Universidade do Minho
Escola de Engenharia

Roberto Carlos Sá Ribeiro

Numerical Simulations on Heterogeneous Systems: dynamic workload and power management

outubro de 2018

Numerical Simulations on Heterogeneous Systems: dynamic workload and power management
Roberto Carlos Sá Ribeiro
UMinho | 2018

Universidade do Minho
Escola de Engenharia

Roberto Carlos Sá Ribeiro

Numerical Simulations on Heterogeneous Systems: dynamic workload and power management

Tese de Doutoramento em Informática

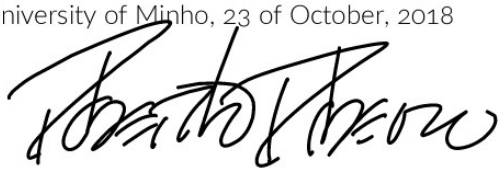
Trabalho efetuado sob a orientação do
Professor Doutor Luís Paulo Santos
do
Professor Doutor Miguel Nóbrega
e do
Professor Doutor Hrvoje Jasak

outubro de 2018

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration. I further declare that I fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, 23 of October, 2018



Acknowledgements

This thesis was not made in one day, and across the challenging days of its development, several were the people that motivated and supported me. I would like to thank my supervising team, Luís Paulo Santos (University of Minho), Miguel Nóbrega (University of Minho) and Hrvoje Jasak (University of Zagreb), for their support and knowledgeful guidance. Among these, I would like to exalt my gratitude to Luís Paulo Santos. By helping me overcoming technical issues, funding issues, motivational issues, among others, I can honestly state that the successful achievement of this work would not be possible without his support.

I also want to thank Professor Alberto Proença (University of Minho), for all the support and especially, for providing the means to pursue and achieve this thesis. I am also truly grateful to João Barbosa (TACC - University of Texas) as he was, not only a friend but also a co-worker, research partner, co-author and brainstorming partner. He was also largely responsible for my time in TACC (Texas, USA), to which I greet and extend my gratitude. Thank you to current and former members of my research group (LabCG, UM), in particular, to Waldir — a brainstorming partner and a friend.

And last but not least, I want to thank my family and friends — as an American writer once said:

> *You can kiss your family and friends good-bye and put miles between you, but at the same time you carry them with you in your heart, your mind, your stomach, because you do not just live in a world but a world lives in you.*

Funding

The work that composes this thesis was funded by National Funds through the *FCT - Fundação para a Ciência e a Tecnologia* (Portuguese Foundation for Science and Technology) and by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) within projects PTDC/EIA-EIA/100035/2008, PEst-OE/EEI/UI0752/2014, FCOMP-01-0124-FEDER-010067 and UID/CTM/50025/2013. Also by the School of Engineering, University of Minho, within project P2SHOCS - Performance Portability on Scalable Heterogeneous Computing Systems and by the PT-FLAD Chair on Smart Cities & Smart Governance. To these entities, I would like to express my sincerest gratitude.

Resources

I would like to thank Kyle Mooney and other authors for providing the code supporting migration of dynamically refined meshes in OpenFOAM (Chapter 4). I would also like to acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin, and the SeARCH computing project at the University of Minho, for providing the HPC resources.

Numerical Simulations on Heterogeneous Systems: dynamic workload and power management

Abstract. Numerical simulations are among the most relevant and computationally demanding applications used by scientists and engineers. As accuracy requirements keep increasing so does the corresponding workload and, consequently, the demand for additional computing power. HPC systems are thus a fundamental tool to allow for a time effective execution of such simulations; performance maximization is therefore a pertinent and crucial subject of research. Over the last decade HPC has undergone a major shift, resulting on heterogeneous parallel computing systems, which integrate devices with different architectures, exposing different instruction sets, programming and execution models, and ultimately, delivering significantly different performances. This heterogeneity raises a variety of challenges to application developers, such as performance and code non-portability, performance imbalances and disjoint memory address spaces. These challenges not only widen the gap between peak and sustained performance, but also significantly reduce development productivity. Additionally, numerical applications often exhibit dynamic workloads, with unpredictable computational requirements, which, together with associated code divergence and branching workflow, further aggravates the heterogeneity challenge — this is defined as the Two-fold Challenge. The increasing scale in HPC systems also leads to a fast growing power consumption, with power management solutions being of crucial importance. The design of such solutions becomes harder within the two-fold challenge context.

This thesis addresses the Two-fold Challenge in the context of numerical simulations and HPC systems, focusing on optimising sustained performance and power consumption. A variety of mechanisms is proposed and validated across different parallel computing paradigms. These mechanisms include a unified execution and programming model, a transparent data management component and heterogeneity-aware dynamic load balancing and power management systems. The contributions of this thesis are divided into three areas: efficient and effective application development and execution on heterogeneous single-nodes with multiple computing devices, load and performance imbalances in heterogeneous distributed systems and power-performance trade-offs in heterogeneous distributed systems. In order to foster the adoption of proposed mechanisms, some were designed and integrated into a widely used numerical simulation library — OpenFOAM. Experimental results assert the effectiveness of the proposed approaches, resulting on significant gains in performance and reduced power consumption in multiple scenarios.

Simulações Numéricas em Sistemas Heterogéneos: carga dinâmica e gestão de potência

Resumo. Simulações numéricas são uma das mais importantes e computacionalmente exigentes aplicações usadas por cientistas e engenheiros. A carga computacional destas aplicações é proporcional aos requisitos de precisão da simulação, que por sua vez, têm aumentado significativamente, resultando numa maior exigência a nível de poder computacional. Os sistemas de computação de alto desempenho (*High Performance Computing* (HPC)) são uma ferramenta fundamental, que permitem executar estas aplicações em tempo útil. Obter o desempenho máximo destes sistemas é portanto uma área de investigação de elevada importância e pertinência. Na ultima década, a computação de alto desempenho tem sido alvo de consideráveis mudanças, resultando em sistemas computacionais paralelos e heterogéneos. Estes sistemas são compostos por dispositivos com diferentes arquiteturas, *instruction sets* e modelos de programação e execução, resultando em desempenhos significativamente diferentes. Esta heterogeneidade levanta vários desafios, nomeadamente, código da aplicação e desempenho não portáveis entre dispositivos, diferenças de desempenho e espaços de endereçamento de memória disjuntos. Estes desafios, não só aumentam a diferença entre o pico de desempenho e o desempenho obtido, mas também reduzem significativamente a produtividade. Mais ainda, as aplicações numéricas exibem, frequentemente, cargas dinâmicas, cujos requisitos computacionais são imprevisíveis. Este dinamismo, combinado com a divergência do código e com o controlo de fluxo condicional, agrava as complexidades associadas à heterogeneidade do sistema, sendo referido como *Two-fold Challenge*. O progressivo aumento da dimensão dos sistemas HPC tem também, como consequência, um rápido aumento do consumo de potência. Sistemas de gestão de potência são portanto de extrema importância, no entanto, o desenvolvimento destes sistemas torna-se complexo perante o *Two-fold Challenge*

Esta tese aborda o *Two-fold Challenge* no contexto de simulações numéricas e sistemas HPC, focando-se na otimização do desempenho e potência consumida. Vários mecanismos são propostos e validados em diferentes paradigmas de computação paralela. Nomeadamente, modelos unificados de execução e programação, sistemas transparentes de gestão de dados e sistemas de balanceamento de carga e gestão de energia baseados na heterogeneidade do sistema. As contribuições desta tese são divididas em três áreas: desenvolvimento e execução eficiente de aplicações em sistemas heterogéneos com um único nó e múltiplos dispositivos, desbalanceamento de carga computacional e desempenho em sistemas heterogéneos distribuídos e compromissos entre desempenho e potência consumida em sistemas heterogéneos distribuídos. De forma a promover o uso dos mecanismos propostos, parte destes foram desenvolvidos e integrados numa conceituada biblioteca de simulações numéricas — OpenFOAM. Resultados experimentais validam a eficácia dos mecanismos propostos, resultando em ganhos significativos de desempenho e redução de potência consumida em múltiplos cenários.

Contents

Acronyms

ACPI Advanced Configuration and Power Interface
AMR Adaptive Mesh Refinement
AS Applied Scientist
BWU Basic Work Unit
CFD Computational Fluid Dynamics
CommGraph Communication Graph
CU Computing Unit
DAG Direct-acyclic Graph
DD Domain Decomposition
DM Decision Model
DMS Data Management System
DLB Dynamic Load Balancing
DVFS Dynamic Voltage and Frequency Scaling
FE Finite Element
FLOPS Floating-point Operations Per Second
FV Finite Volume
FPGA Field-programmable gate array
GFLOPS Giga Floating-point Operations Per Second
GPGPU General Purpose GPU
HEFT Heterogeneous Earliest Finish Time
HPC High Performance Computing
HDS Heterogeneous Distributed Systems
HSNS Heterogeneous Single-Node Systems
HS Heterogeneous Systems
DW Dynamic Workload
DSP Digital Signal Processor
ILP Instruction Level Paralellism
KNL Knights Landing
MPI Message Passaging Interface
OPM Online Profiling Module
OpenFOAM Open Source Field Operation and Manipulation
PAS Power-Adaptive Scheduler

PM Performance Model

RHeAPAS Runtime Heterogeneity-Aware Power-Adaptive Scheduler

RM Repartitioning Module

RSD Relative Standard Deviation

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Threads

TDP Thermal Design Power

TPL Task Parallel Library

UDP Uniform Distribution of Power

List of Figures

List of Tables

List of Infos

Chapter

1 Introduction

Contents

Civilization is a progress from an indefinite, incoherent homogeneity toward a definite, coherent heterogeneity.

Herbert Spencer

In this initial chapter, a detailed discussion on the context and motivation of this thesis is provided. An overview of the state of today's computing systems is presented, together with the identification of a set of challenges that emerged from the way computing technologies evolved. It also introduces a hierarchical taxonomy that categorizes the paradigms of parallel computing. In the final sections, a brief overview of how this thesis will approach the identified challenges is provided, defining the research hypothesis, goals and contributions of this thesis.

1.1 Context and Motivation

Numerical computer simulations extend the human capability to acquire knowledge on fundamental aspects of physics. They allow studying the behaviour of complex physical systems that are impractical to assess either by analytical or experimental methods. Weather forecasting, financial market forecasting, medical and health-care development, image synthesis and rendering, aircraft aerodynamics are just some of the examples of a vast set of applications that make use of numerical simulations. The value and accuracy of these simulations are correlated with their workload which has a direct impact on the computational effort. For instance, larger refined models of a city in weather forecasting yield a more accurate and wider weather prediction, a larger pixel sampling and resolution in a physically based rendering algorithm results in higher-quality photo-realistic images. This results in a generalized demand for computing power in order to enable higher quality and complex simulations [1].

> **Numerical Simulation by *nature.com***
>
> *A numerical simulation is a calculation that is run on a computer following a program that implements a mathematical model for a physical system. Numerical simulations are required to study the behaviour of systems whose mathematical models are too complex to provide analytical solutions, as in most non-linear systems.*

Given the computing complexity and volume of the data associated with these applications, their execution times easily achieve the order of days or weeks, largely depending on the computing resources available. High Performance Computing (HPC) systems are a fundamental tool used by scientists and engineers to implement and run numerical simulations as they offer higher levels of computing capabilities. HPC systems performance delivery is fundamentally based on parallel computing and scalability provided by stand-alone sophisticated servers and by large-scale clusters with thousands of compute nodes. Maximizing the extracted performance from these systems is, therefore, a pertinent and crucial subject of research and development.

In the last decades, driven by the surge in computational requirements, HPC systems have been subject to significant change in architectural design and development. Manufacturers consistently increased the CPU transistor count and devised sophisticated approaches to organize chip space in order to further increase performance (e.g. vectorization, superscalar, etc). However, limited by the power wall, chip

Clusters, Supercomputers and HPC Systems

Cluster is a core architectural concept based on a set of computers connected by a local network providing extended parallel computing capabilities. **Supercomputer** is a large-scale computing infrastructure based on the cluster concept and used in the solving of complex and large scientific problems. **HPC systems** are computing platforms targeted to deliver higher levels of computational power aimed to solve complex computational problems. HPC systems range from standalone compute servers to large-scale supercomputers.

manufacturers adopted new architectures and embraced parallelism within the chip as the mainstream approach to overcome the physical limitations [2], [3]. Multi-core CPUs quickly became ubiquitous as well as sophisticated computing paradigms such as multi-threading.

With the introduction and development of programmable shaders, along with advancements in floating-point support, GPUs became able to compute highly-parallel scientific tasks with substantially higher performance than common CPUs due to its many-core architecture. With the introduction of highly productive programming models, such as CUDA and OpenCL, these co-processing devices became general purpose scientific computing devices and ubiquitous across HPC systems. Intel also pursued the many-core co-processing approach with the first Intel Xeon PHI device that consisted in a 48-core chip with wide-SIMD capabilities and x86 compatible. Intel continued to develop these micro-architectures and recently released a new many-core self-hosted [1] device with codename Knights Landing (KNL) with 64 cores.

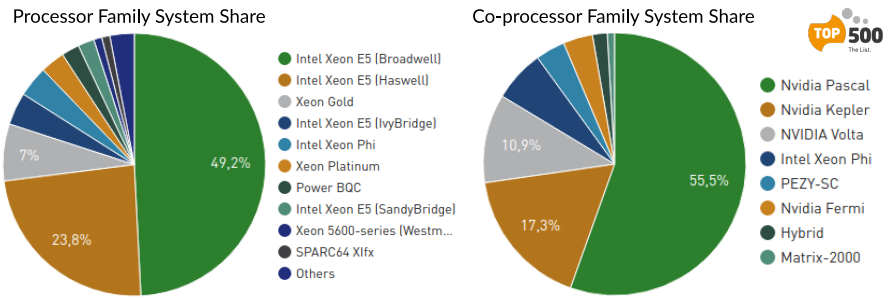


Figure 1.1: Processor and co-processor family system share from June 2018 Top500 supercomputer list

Sophisticated multi-core CPUs, many-core GPUs and Xeon PHI's are common devices equipped in standalone HPC compute servers and supercomputer nodes. This multitude of devices fundamentally revolutionized the plurality in terms of HPC technology, outsetting the ***Heterogeneous Computing Era*** [4]. This is clearly observed in Figure 1.1 that illustrates the processor and co-processor family systems share in the Top500 supercomputer list. Compute nodes are heterogeneous as they are composed of devices that are designed with different architectures, using different instruction sets, programming and execution models, and ultimately deliver significantly different performances. Clusters are rendered heterogeneous as

[1] the many-core is the main CPU

they can be easily extended with nodes with more efficient CPU architectures and new sophisticated co-processing devices. Figure 1.2 briefly illustrates some examples of heterogeneous supercomputers and their architectures present in the top 60 places of the Top500 supercomputers list[1]. For instance, the *Pleiades* from NASA, is composed by four different Intel architectures and a NVIDIA GPU architecture.

Supercomputer	Architectures
Stampede2	Skylake, KNL
Marconi	Broadwell, KNL
Pleiades	Broadwell, Haswell, Ivy Bridge, Sandy Bridge, Tesla K40
Jureca	Haswell, KNL, Tesla K40
Electra	Skylake, Broadwell
Thunder	Haswell E5-2699v3, Haswell E5-2697v3, KNC, Tesla K40
Mistral	Broadwell, Haswell, Tesla K80, Quadro M6000

Figure 1.2: Some examples of heterogeneous supercomputers in the top 60 places of the June 2018 Top500 Supercomputing list.

1.1.1 4-Tier Parallelism

With the adoption of parallelism as the mainstream paradigm to increase performance and considering the current structure of HPC systems, a hierarchical parallel computing structure can be defined using a 4-Tier hierarchical taxonomy:

- **Tier-4** Inter-node parallelism in a distributed memory system where nodes compute concurrently. Each node has its own memory addressing space.
- **Tier-3** Intra-node parallelism within a single compute node with multiple compute devices, also known as inter-device parallelism or in some cases hybrid execution where devices (e.g CPU and GPU) compute concurrently. Typically each device has its own memory space.
- **Tier-2** Intra-device parallelism, parallelism within device where cores compute concurrently. Memory space is typically shared across cores.
- **Tier-1** Intra-core parallelism, a typical example is Single Instruction Multiple Data (SIMD) execution also known as vectorization. In some devices, dedicated registers are used. GPUs also promote SIMD execution using a massive multi-threading approach.

A **Tier-0** could be defined as the Instruction Level Paralellism (ILP) within core considering superscalar execution and instruction pipelining.

[1] Note that some heterogeneous supercomputers may have some restriction policies that limit the use of different architectures simultaneously.

1.1.2 Heterogeneous Systems and Dynamic workloads

With the embracing of parallelism and with the advent of the heterogeneous era, several challenges emerge that not only affect the application development productivity but also severely impact the extracted performance of today's computing systems. These challenges are further aggravated in the presence of dynamic workload applications such as numerical simulations. This section briefly describes the challenges posed by parallel heterogeneous systems and in particular the impact of combining them with dynamic workloads.

The Heterogeneity Challenge

A cluster can be fairly easily extended by adding more compute nodes with similar architectures, but often from newer and more sophisticated generations offering more computing capabilities. This same extensibility, however, renders the system heterogeneous in the sense that different generations of hardware with different levels of performance coexist across nodes leading to **performance imbalances**. These performance imbalances are also originated from the diversity of devices that constitute a node. Resource idling and underutilization along with poor scalability are the major consequences of an imbalanced system [5], [6].

In the presence of multiple devices with different architectures, one of the major challenges is the **performance non-portability** across devices. For instance, an application optimized for the CPU may deliver far less performance in a GPU and vice-versa [7]. This is due to the different execution models and associated device architecture details that are designed to address different types of workloads. Programmers need to re-design their applications in order fully benefit from each device computing capabilities.

In devices with **disjoint memory address spaces** – such as GPUs and other co-processors – application data must travel through a limited bandwidth bus (PCI-Express), which results in a potential performance bottleneck. Data transfers must be explicitly managed and minimized for consistency and efficiency purposes[8]. This not only affects performance, but significantly reduces productivity. Moreover, these devices are typically used with libraries and programming tools (CUDA, OpenMP, Intel TBB, etc) developed by each of the different manufacturers that reflect the differences in execution models and architectures of their devices. Programmers need to comply with these divergent programming models resulting in **non-portable code**.

Dynamic Workload Challenge

Most numerical computer simulations are data-parallel. Data-parallel applications distribute data to compute units that apply some computational operation (or kernel) on the assigned data in parallel. Data is typically defined by a set of work units that represent some entity, object or modelling element, for

instance, a cell of a discretized domain in Finite Volume (FV) Computational Fluid Dynamics (CFD) simulations [9].

Data-parallel applications can be classified in two types — static or dynamic. In **static** (also known as regular) applications, the workload is the same for the entire execution. The number of work units is known *a priori*, typically defined in the start of the application and divided across computing resources or submitted to a queue for processing. These applications exhibit a constant and predictable computational effort requiring simpler scheduling and partitioning heuristics in order to be efficiently distributed across computational resources.

With **dynamic** applications, each of the data elements can be subdivided, merged or generate more work units, rendering the computational effort unpredictable and irregular [10], [11]. The distribution of dynamic workloads across parallel computing resources becomes a far more complex challenge due to an unpredictable number of work units and/or an unknown number of operations per work unit. If an uniform distribution of workload is applied, each compute unit will receive the same amount of work units. However, since each work unit may require an **arbitrary amount of computational effort**, the system will be imbalanced and resource idling occurs.

A **sub-type** of dynamic applications can be defined as **irregular applications** when the generation of work results in **code divergence and branching workflow**. These applications will significantly hinder the performance in many-core devices, such as the GPU [12]. The massively threaded execution model favours well defined and regular code, but with irregular workloads, arbitrary (uncoalesced) memory accesses and unpredictable complex execution patterns will potentially result in significant performance losses.

Application workload types: static, regular, dynamic and irregular

Data-parallel applications can be classified in two main types — static or dynamic. **Static applications,** also known as **regular applications**, exhibit a constant and predictable computational workload across the whole compute time. The number of work units is known *a priori*, typically defined in the start of the application and divided across computing resources or submitted to a queue for processing. Examples of static data-parallel applications are matrix multiplications and decompositions, where the number of elements is known and the operations per element can be determined and thus so the global number of operations. CFD simulations with static meshes can also be considered static applications as they require a uniform computational effort along the runtime.

With **dynamic applications** the workload is generally unpredictable and irregular across the runtime. Each of the data elements can be subdivided, merged or generate more work units resulting in an unpredictable amount of computational effort associated with each of the work units. For instance, a CFD simulation with Adaptive Mesh Refinement is considered a dynamic application since each cell can be recursively subdivided or merged along the simulation depending on fluid flow or other properties. A sub-type of dynamic applications can be defined as **irregular applications** when the generation of work results in code divergence and branching workflow. These applications are typically characterized by irregular data structures, irregular control flow and/or irregular communication patterns with uncoalesced memory accesses. An example of an irregular application is the Monte-Carlo physically based rendering engine. The workload associated with processing a pixel is unpredictable since both direction and length of the path of the pixel ray are stochastically generated and scene dependent.

Yet, dynamic data-parallel applications constitute the largest percentage of numerical computer simulations, not only because they are typically associated with complex real-world data and models but also because they are expressed using irregular algorithms such as random walks [13], [14], graph and sparse matrix algorithms [15], [16], particle simulations [17]–[19], meshing techniques [9], [20], among others. The pertinence of these applications renders the study and development of workload scheduling algorithms crucial.

The *Two-fold Challenge*

As discussed in the two previous sections, Applied Scientists (ASs) rely on HPC systems to perform numerical simulations. These systems, however, are heterogeneous and pose a number of challenges that need to be addressed in order to be efficiently used. Furthermore, numerical simulations are prone to exhibit a dynamic and unpredictable workload behaviour that is hard to be efficiently distributed and executed. The combination of these two computing features results in a further enlargement of the complexity of the individual challenges identified — this is defined as the ***Two-fold Challenge***.

The workload that needs to be scheduled is now dynamic and unpredictable, which aggravates the performance imbalance issue among the heterogeneous computing units. Computing units with less performance and already causing imbalance may sustain a workload increase which will substantially increase the idling of faster units. Dynamic generation of work will also promote code divergence and branching which aggravates the performance portability issue. In the presence of multiple devices, accounting for different execution models becomes a more complex task when computing divergent and branched workflow. Data management becomes also non-trivial since the data required by devices is potentially arbitrary.

Indeed, these two computing features – resource heterogeneity and dynamic workload – are relevant topics but boost each other and correlate in hindering productivity and performance extraction. Notice that all these challenges are addressed either by a computer scientist or an AS. The latter is a non-expert programmer that usually has basic programming skills and computer science knowledge. Design and development of mechanisms to counter the challenges posed by the combination of these features, specifically, in relevant applications like numerical simulations, is, therefore, a pertinent research area.

1.1.3 Power Consumption Challenge

The increasing scale of HPC systems leads to a **fast-growing power consumption** that is becoming one of the major concerns in developing and maintaining these systems [21]. The cost of energy required to power a supercomputer tends to surpass the cost of the system itself, resulting in a huge economic impact but also the inherent consequences in terms of environment. Power management becomes of crucial importance where HPC solutions – either hardware and software – need to be re-evaluated in

terms of power-efficiency [22]. Since computing devices are based on electrical integrated circuits, power consumption has a close correlation to performance. A power management system must seek to reduce power consumption but also maintain acceptable levels of performance.

However, **power management becomes a far more difficult** challenge in systems exposed to the two-fold challenge. Each of the multiple devices that co-exist in a system may exhibit different power consumptions and different performances. Any strategy to reduce power consumption becomes non-trivial where the power manager needs to account for the impact of power changes and the subsequent impact on performance that is particular to each device. In the presence of dynamic workloads, designing of such strategy becomes even more complex given the unpredictability of the workload generated by the application.

1.2 Facing the Challenges

Performance imbalances are caused by a plurality of devices and architectures and by the dynamic nature of the workload associated with many applications, such as the numerical simulations addressed throughout this thesis. These issues can be addressed by rising awareness on each Computing Unit (CU) performance using **performance models** and combine that information with runtime **Dynamic Load Balancing (DLB)**. These features will provide heterogeneity-aware workload partitioning and redistribution that will assign and migrate work according to performances and current system load. This will minimize resource idling thus increasing utilization and scalability.

> **Computing Unit (CU)**
>
> A Computing Unit is an abstraction used in this document that represents a device or a set of devices that perform computation. For instance, a single CPU core, a CPU, a GPU, a cluster compute node, etc.

The diversity of devices' computing models also causes the application implementation and optimization to be non-portable. Both performance optimizations and code implementation can not be efficiently and transparently applied to multiple different devices. In order to address this issue, an **unified execution and programming model** can be proposed. The unified execution model will account for the different particularities of each device and provide an execution workflow that is both transparent to the user and accounts for the code divergence and irregular workflow of dynamic applications. This will increase productivity and will try to improve the performance of devices that do not favour dynamic and irregular applications. The unified programming model should be device agnostic and hide code primitive details from the programmer, fundamentally increasing productivity.

The disjoint memory address spaces of co-processors results in explicit data management for consistency and efficiency purposes. By designing and integrating a **data management system**, transparent data transfers can be performed and optimization mechanisms such as locality-aware scheduling can be

applied. Data management is accessed by an API that is part of the unified programming model, therefore providing device agnostic data management and further increasing productivity.

Power consumption is one of the most concerning aspects in today's computing systems. Heterogeneous systems and dynamic workloads further hamper the power management challenge. A **dynamic and adaptive heterogeneity-aware power assignment** is thus required that will account for dynamic changes of the workload and perform power assignment decisions while weighing performance impact. The power decisions can be supported by a **performance model combined with a power model** resulting in a unified power-performance efficiency mechanism.

1.3 Goals and Contributions

The research hypothesis put forward by this thesis is that the challenges raised across the 4-Tiers of parallelism by the heterogeneity of resources, the dynamic nature of the computational workload and the huge power consumption of current HPC systems can be effectively addressed by a thoughtful combination of the above described mechanisms. In particular:

- a **unified execution and programming model** for heterogeneous systems, fully integrated with a transparent **data management system**, will effectively address the **performance portability** challenge, while simultaneously **increasing programming productivity** and promoting utilization of HS among AS;
- **dynamic load balancing** and heterogeneity aware scheduling, properly grounded on robust and light weight **performance models**, will address the above identified two-fold challenge, optimizing resource utilization and orchestration towards minimization of application execution time;
- appropriate **heterogeneity aware power management mechanisms** can effectively limit power consumption while increasing performance when compared with an uniform distribution of the available power budget.

The major goal of this thesis is thus to design, integrate and assess these techniques and provide tools to efficiently and productively develop numerical computer simulations in state of the art HPC systems. The contributions of this thesis cover different systems combined with different applications and address their issues in a scientific and engineering perspective by improving and integrating existing techniques. The following section briefly introduces the main contributions and a detailed discussion is provided in the following chapters.

1.3.1 Main Contributions

The contributions of this thesis are divided in three main areas, each targeting a different tier or set of tiers. Each area tries to address a set of challenges that are common to a particular goal and system resource configuration. The main contributions of this thesis are as follows:

Handling Heterogeneous Single-Node Systems (HSNS) The challenges in single node multi-device systems (Tier-3, 2 and 1) are addressed by proposing a unified task-based programming and execution model tailored to efficiently execute data-parallel regular and irregular applications. The integration of persistent kernels is proposed as an intra-device scheduling mechanism along with transparent data partitioning and a device agnostic programming model. The proposed mechanisms are implemented and evaluated with multiple applications and various configurations of CPUs and GPUs. A direct comparison to a state-of-the-art framework is also performed.

Contribution I

This contribution that is supported by a scientific paper published in the *Parallel Processing Letters* journal published by *World Scientific*.
R. Ribeiro, J. Barbosa, and L. P. Santos, "A Framework for Efficient Execution of Data Parallel Irregular Applications on Heterogeneous Systems", Parallel Processing Letters, vol. 25, no. 2, p. 1550004, Jun. 2015. DOI:10.1142/S0129626415500048

Runtime heterogeneous-aware load manager for Heterogeneous Distributed Systems (HDS) This contribution evaluates the combination of a DLB system with an application-oriented performance model as a mean to increase resource utilization in performance and workload imbalanced systems. The contribution targets distributed-memory systems (Tier-4) and the designed approach is directly integrated and evaluated in a widely used CFD library (OpenFOAM). It is based on a definition of a Performance Model combined with a decision model that performs educated decisions on how to assign data parallel workload, converging to a balanced computational effort and thus increasing resource utilization. Evaluation is performed across multiple combinations of static and dynamic workload with homogeneous and heterogeneous resource configurations.

Contribution II

This contribution is supported by a scientific paper published in the proceedings of the conference *International Conference on Computational Science - ICCS 2018*, part of the *Lecture Notes in Computer Science* book series by *Springer*.
R. Ribeiro, L. P. Santos, and J. M. Nóbrega, "nSharma: Numerical Simulation Heterogeneity Aware Runtime Manager for OpenFOAM", in Lecture Notes in Computer Science, Springer International Publishing, 2018, pp. 429–443, volume 10860. DOI:10.1007/978-3-319-93698-7_33

Runtime heterogeneous-aware power-adaptive scheduler for HDS Power management is expressed as an optimization problem in order to improve power efficiency and performance in power-limited

scenarios. The proposed model is formulated based on two merged objectives: power consumption minimization and performance maximization. Heterogeneity awareness is provided by a performance model and power assignment decisions are adaptively performed at runtime. The approach is evaluated with CFD simulations with dynamic workload running on HDS (Tier-4 parallelism, e.g. multiple CPU generations and KNL nodes). Power consumption reduction and performance behaviour are discussed as well as assessments on energy consumption.

Contribution III

This contribution that is supported by a scientific paper published in the proceedings of the conference *International Conference on High Performance Computing & Simulation - HPCS 2018*.
R. Ribeiro, L. P. Santos, and J. M. Nóbrega, "Runtime heterogeneous-aware power-adaptive scheduling in OpenFOAM", in 2018 International Conference on High Performance Computing & Simulation (HPCS), 2018.

1.3.2 Experimental Context

This work's hypothesis, goal and contributions are clearly presented throughout Section 1.3. The focus is on heterogeneous parallel computing systems and on the efficient and productive development and execution of numerical computer simulations. There is, however, a major technological shift on the experimental contexts used to validate the proposed hypothesis between the first and the remaining two contributions as identified in Section 1.3.1. The former proposes a specific framework for the development and execution of irregular applications on heterogeneous systems. This framework was conceived and developed within the context of this thesis and requires applications to be developed in compliance with the proposed programming and execution model. The latter contributions propose two plugins written in C++ that integrate onto OpenFOAM in a transparent manner for the OpenFOAM application developer. The reason for this shift in the experimental approach is very pragmatic. As the work progressed from its very early initial stages the team engaged on a collaboration with the University's Institute of Polymers, whose researchers often use OpenFOAM over parallel systems to solve CFD-related problems. It was felt by all that **this thesis' results could be useful for this community**, in particular if the proposed techniques could be applied in a straightforward (eventually transparent) manner. A decision was therefore made to adopt OpenFOAM as the experimental use case.

OpenFOAM is a large and complex CFD simulation framework, with extensions and plugins developed on an open source approach by many practitioners, widely distributed both geographically and institutionally. Adapting OpenFOAM (or a subset) to the initially proposed heterogeneous framework would be an unfeasible task. The reason for this is essentially based on some of the principles that defined the initial framework. These principles resulted in specific application requirements that OpenFOAM does not meet out-of-the-box (such as loosely-coupled data-parallel execution). On the other hand, developing

OpenFOAM specific plugins (as is the case of nSharma and RHeAPAS, see Chapters 4 and 5) was deemed feasible, although complex, and their seamless integration with OpenFOAM promotes their adoption by OpenFOAM application developers. This was therefore the path followed throughout this thesis second and third contributions. The author believes that this technological choice has no impact on the scientific validity of the presented findings, with the added benefit of facilitating knowledge transfer from computer science researchers to parallel CFD simulations users.

1.4 Thesis Structure

This thesis document is organized in six chapters, two for introductory content and background, three for main contributions and a final concluding chapter.

Chapter 1 — Introduction This chapter provides the context and motivation of this thesis and identifies some of the challenges posed by heterogeneous systems. It also defines the thesis hypothesis and outlines its contributions.

Chapter 2 — Background An overview of hardware and software standard solutions is discussed, including a straightforward categorization of modern HPC architectures, followed by the most commonly used APIs and developing tools to work with them. The final section discusses the main issues with these technologies in the context of heterogeneous parallel systems and how they can be addressed.

Chapter 3 — Heterogeneous Single-node Systems This chapter describes the first contribution of this thesis where an approach to address the challenges emerged from single-node heterogeneous parallel systems are addressed.

Chapter 4 — Heterogeneous Distributed Systems An approach to tackle the challenges posed by multi-node heterogeneous systems is proposed. The proposed mechanisms are essentially based on a dynamic load balancing technique, designed to handle dynamic workloads in systems with performance imbalances across computing nodes.

Chapter 5 — Power Scheduling in Heterogeneous Distributed Systems This presents the third and last contribution, focusing on the power management challenges of heterogeneous distributed systems. It proposed a heterogeneity-aware power-adaptive scheduler based on the solving of an optimization problem. It is recommended to read the Chapter 4 before this chapter.

Chapter 6 — Conclusions and Future work General conclusions are provided asserting the successful validation of the thesis hypothesis. The future work is also discussed, where a new model is proposed for development and assessment.

Chapter

2 Background

Contents

This chapter provides a brief overview of hardware and software architectures, including standards and manufacturer tools. It provides a straightforward categorization of modern HPC architectures, followed by the most commonly used APIs and developing tools to work with them. The focus this chapter is on technology that is actually used in today's systems. The final section discusses the main issues with these technologies in the context of heterogeneous parallel systems and how they can be addressed. Detailed related work will be discussed in each of the contribution chapters.

2.1 Modern HPC Architectures

Modern HPC systems are composed by a plurality of devices that can be categorized in three main architectures: multi-core CPUs, many-core CPUs and many-core co-processors (which include GPUs). These devices are used both by single node HPC systems or across nodes of a cluster. Single node systems are typically composed by one or more multi-core CPUs and in multiple cases, a high number of co-processors (e.g. NVIDIA DGX-1 with 8 GPUs). The nodes that compose any of the systems in the Top500 [23] list are composed of one or more devices from one or more of these categories.

2.1.1 Multi-core CPUs

Multi-core CPUs are designed to be as general purpose as possible. Manufacturers try to develop and enhance CPUs based on complex trade-offs in order to efficiently compute the widest possible range of applications. This results on a chip endowed with extremely complex features but, as a consequence, limited parallelism. Looking at the list of supercomputers, Intel has the larger processor share followed by IBM (PowerPC) and Fujitsu (SPARC). The most recent Intel micro-architecture already used in some systems is codenamed *Skylake*.

Skylake chips were introduced in mid-2017, built with 14nm with a core count from 4 to 28 [1] with Intel's Hyper-threading technology resulting on 8 to 56 virtual processors. With base operating frequencies ranging from 1.6 GHz to 3.7 GHz and a Thermal Design Power (TDP) between 60W and 240W[2], these devices perform out-of-order execution with 14 to 19 pipelining stages, branch-prediction, speculative and superscalar execution. The chip also includes three levels of associative cache with more than 1MB per core for the second and third level. Each core is equipped with multiple scalar and vector arithmetic units that provide SIMD operations, which in this latest architecture version has been extended to 512-bit registers (AVX-512). In terms of theoretical performance, the Skylake based Intel Xeon Platinum 8160, for instance, has peak double precision of about **1612 Giga Floating-point Operations Per Second (GFLOPS)**

[1] typically in high-end servers such as supercomputer nodes, chip versions with 24 to 28 cores are used
[2] high-end versions have an average of 160W

[1].

To get an insight on the performance differences across older multi-core CPUs that still coexist in the same system, the NASA *Electra* supercomputer, for instance, is composed of multiple *Skylake* nodes together with *Broadwell* nodes. The *Broadwell* nodes are composed of Intel Xeon *Broadwell* E5-2680 v4 CPUs (launched in Q1 2016) with 14 cores, 2.4 GHz of base frequency and two 256 bit arithmetic vector units, resulting in **537,6 GFLOPS**[2].

2.1.2 Many-core CPUs and Co-processors

Intel Xeon Phi, formerly known as Intel Many Integrated Core, is a family of x86-compatible many-core devices targeting high-performance massively parallel computing by devoting more transistors to a higher number of simpler cores.

The first production model, still present in multiple HPC systems, is an external device connected to the main system by a PCI-Express bus. Its micro-architecture is codenamed Knights Corner and provides a core count ranging from 57 to 61 cores with an hyper-threading of 4 and from 6 to 16GB of dedicated memory. They operate between 1.0 to 1.2GHz of base clock frequency with most versions exhibiting a TDP of 300W. Cores are connected using a ring topology and each core is based on a modified version of an Intel Pentium Core with two levels of cache and 512-bit vector operations. The theoretical peak performance of a Intel Xeon Phi SE10P is **1073 GFLOPS**[3].

The second generation of the Xeon Phi architecture is codenamed Knights Landing and was deployed as an external board but also as a standalone self-hosted CPU. They are still targeted for massively parallel computing however they can be configured without any other main device. These devices pack a slight increase in core count from 64 to 72 cores and also an increase in base frequency delivering 1.3 to 1.5GHz of clock speed. The core arrangement is slightly more sophisticated where the modified Intel Atom based cores are organized in tiles interconnected by a 2D mesh. The chip also contains 8 new banks of high bandwidth memory – known as Multi-Channel DRAM (MCDRAM) – of 16GB each. Communication approach between tiles and the use of the MCDRAM can be configured at boot time with different modes that introduce some flexibility in exploiting chip performance. The theoretical peak performance of a Intel Xeon Phi 7250 is **3046 GFLOPS**[4].

[1] 2.1 (GHz) x 24 (cores) x 512/64 (DP AVX) x 2 (FMA units) x 2 (FMA); frequency of AVX units is variable, so actual theoretical performance may be slightly different
[2] 2.4 (GHz) x 14 (cores) x 256/64 (DP AVX) x 2 (FMA units) x 2 (FMA))
[3] 1.1 (GHz) x 61 (cores) x 512/64 (DP AVX) x 1 (FMA unit) x 2 (FMA)
[4] 1.4 (GHz) x 68 (cores) x 512/64 (DP AVX) x 2 (FMA units) x 2 (FMA)

2.1.3 GPUs

GPUs dominate the share in Top500 co-processors with NVIDIA as the main manufacturer. NVIDIA GPUs are mostly external devices connected through PCI-Express bus and dedicated memory. Most of the transistors are devoted to data-parallelism providing a Single Instruction Multiple Threads (SIMT) execution and programming model.

The chip is composed of a set of multiprocessors that create, schedule and execute groups of threads called *warps*. Each multiprocessor contains multiple execution cores and special function units that will concurrently execute the instructions of the warp. The programming model defines a grid of threads which is divided into blocks which in turn are internally organized into warps [1]. In a typical implementation of a GPU application, each thread is associated with a data-parallel work unit and is then executed in an instruction lock-step with the other threads in the warp[2] in a SIMD way. This architecture differs from vector processing in the sense that each thread execute its own instruction allowing programmers to write thread-level parallel code for independent threads[3] [24].

Contrary to multi-core CPUs, there is no branch-prediction nor speculative execution, these devices are designed for maximum throughput by efficiently managing thousands of threads and resorting to memory latency hiding mechanisms (e.g. fast context switching). The latest most commonly available versions of NVIDIA chips are based on the Pascal micro-architecture. For instance, the Tesla P100 has a total of 3584 (simple) cores, 16GB of dedicated memory and operating at a base frequency of 1.2GHz resulting in a TDP of 250W. According to the manufacturer, it has **5300 GFLOPS** of peak double precision performance.

These are the most commonly used devices in HPC but several others architectures coexist contributing to the heterogeneous ubiquity, such as AMD multi-core CPUs, AMD GPUs, Xilinx and Altera Field-programmable gate arrays (FPGAs), Texas Instruments Digital Signal Processors (DSPs), among others.

2.2 Parallel Programming Models

Shared memory and distributed memory are two base model abstractions commonly referred to in parallel programming [25]. **Shared memory** allows multiple computing units to access the same memory space, using it for communication purposes and data sharing. This model is typically used with multi-core and many-core CPUs along with multi-threaded programming where each thread has access to node system memory. Since data is shared across threads, data consistency is maintained by the programmer using

[1] warps are a hardware scheduling unit, not part of the programming model

[2] in recent architecture, like NVIDIA Volta, independent thread scheduling is allowed where a program counter and call stack are maintained per thread.

[3] a thread within warp with a different instruction from the other threads will diverge and execute its instructions while the others wait

data concurrency primitives provided by the programming tools.

Distributed memory is typically associated with clusters where each of the nodes has its own physically separated memory space. Communications are performed explicitly using programming primitives that transfer data between nodes using a communication protocol. Since nodes are connected by a network, all data transfers and synchronization signals travel through the network. The distributed memory concept may also be applied to multiple GPUs and other co-processors on the same node, each having its own memory space. Communications are typically performed over a PCI-Express bus and it is the programmer responsibility to ensure data consistency and synchronization orchestration. However, for simplification purposes, in the scope of this document distributed memory systems will always refer to clusters with network node inter-connections.

The following sections provide a brief overview of the APIs that are considered of standard and wide use when developing parallel numerical simulations in HPC systems. These development tools originate either from standards defined by committees of major hardware and software vendors or proposed by individual ones in order to use their devices.

Parallel APIs

In distributed memory systems, the **MPI** [26] standard is the main-stream tool to develop parallel applications. MPI defines an API that allows for orchestration and communication between processes that are hosted in different nodes and/or in the same node. The API provides point-to-point and collective communication primitives that essentially include data transfers and synchronization mechanisms. It is designed basically for the distributed memory paradigm where each MPI process (also known as rank) has its own memory space and it is responsible to handle its own data and execution flow. In a pure-MPI application, each MPI rank will be bond to a core resulting in multiple ranks per node. This mapping can be performed automatically or explicitly controlled by the programmer using process affinities. MPI can also be used in order to perform concurrent execution among multi-core CPUs and many-core co-processors. The Xeon Phi Knights Corner runtime time system allows the MPI library to launch processes allocated within the device and run applications concurrently in a distributed memory approach.

With multi-core and many-core devices, the shared memory approach is typically used and combined with multi-threaded processing. High-level APIs such as **OpenMP** [27] are widely used where the programmer by the means of compiler directives can specify portions of code that are due to run in parallel. OpenMP is then responsible to transparently create and manage threads using a master-slave threading approach. The API will create a specified or automatically detected number of threads and assign each one to a core[1]. **Intel TBB** [28] is a more recent shared-memory library with an increasing adoption that provides

[1] thread-core affinity can also be specified

a more robust and feature-rich parallel library. It provides concurrent data structures, synchronization features, task scheduling, among others. **Cilk Plus** [29] and **Microsoft Task Parallel Library (TPL)** [30] are similar libraries, however, Cilk Plus support has been recently deprecated by Intel and TPL is specific to Microsoft .NET technologies. Lower level APIs can also be used to perform thread level parallelism such as **PThreads** [31], **Boost C++** [32] and **C++11 Standard Libraries**[1] [33].

NVIDIA GPU applications are typically developed using the computing platform **CUDA** [24] which provides a compiler, runtime API and other developments tools. The application is developed using extensions to C/C++ where the programmer defines data transfer policies and functional routines – known as kernels – to be executed by the device and a computing resource requirement specification – all using explicit code tags. A kernel call will create a grid that is subdivided into blocks of threads, both grid and a block can be organized in 1,2 and 3D abstractions. Each thread has a local memory and each thread block has an on-chip shared memory space accessible to all threads in the scope of the block. All threads have access to the device global memory.

GPUs as co-processors have their own memory, which means data to be computed must be migrated from host to device memory through PCI-Express channels. Up until the latest Pascal micro-architecture, data consistency and memory fault between host and device was explicitly managed by the programmer. With the advent of the Pascal architecture in 2016, a transparent memory page-fault system was introduced providing automatic data consistency and migration between host and devices.

These programming models can be combined together providing full hybrid computing across multiple tiers. Each MPI process can perform multi-threaded parallel execution by using any available threading API. It can also be responsible to host a CUDA application and offload data and computation to the device. For instance, in a cluster composed of N nodes and each node composed by a multi-core CPU and a GPU. The programmer can instruct the MPI library to create N ranks, one per each node, where each rank will use OpenMP to perform multi-threaded processing within the corresponding node and use CUDA to offload computation to the corresponding GPU. Each rank will create a thread per core for the multi-core CPU and issue kernel executions and data transfers to the GPU [34]. In this scenario, the programmer is responsible to write the structure and coordination of all the execution flow and data management.

Programming Models for Heterogeneous Systems

In a combined effort between multiple hardware and software vendors, **OpenCL** standard [35] was proposed in an attempt to develop a unified API able to support multiple parallel computing devices. OpenCL is based on C and its execution and programming model is similar to CUDA's. Using a host-device plat-

[1] C++11 and above

form approach, the API resorts to command-queues that issue data transfers and kernel executions to available devices as well as synchronization primitives. Work assignment granularities and device orchestration are explicitly defined by the programmer. Data management depends on device type, with most GPUs requiring programmer explicit management. It supports NVIDIA GPUs, AMD GPUs, multi-core CPU and co-processors, among other devices. However, OpenCL is just a standard and manufacturers provide their own implementation based on the corresponding device[1] and not always provide full compliance.

Other programming standards have been proposed with the same goal, such as the **OpenACC**[2] [36] that has a similar programming model to OpenMP also supporting computation offload to devices. Both OpenCL and OpenACC are restricted to single node systems.

2.3 Power Management

The power consumption of a processor, W, can be modelled by two components: static and dynamic power dissipation. Static power dissipation depends on voltage and leakage current and it occurs regardless of system activity — for this reason, dynamic power will be considered in this work as the main source of power dissipation and consumption. The dynamic power dissipated can be modelled as:

$$W \propto C \times v^2 \times f \tag{2.1}$$

where C is the capacitance being switched per cycle, v is the supplied voltage and f is the operating frequency [37]. C is constant, so both frequency and voltage affect the power consumed. Frequency and voltage are strictly correlated. The frequency will define the maximum voltage required to operate (lower frequencies require lower voltages), on the other hand, reducing the voltage will reduce the maximum frequency allowed. Power consumption is also related with multiple architectural details and configurations such as the number active cores, thread placement, reduced switch activity, etc [37]. It can be controlled using multiple techniques such as **Dynamic Voltage and Frequency Scaling (DVFS)**, thread packing, dynamic concurrency throttling, among others that can be used together in a synergistic way to reduce consumption[38].

Since frequency and voltage are two of the most influencing factors in power consumption, DVFS mechanisms have been widely used to tackle power consumption. In a multi-core CPU, operating frequency and voltage are changed and accessed using a kernel driver (e.g. *acpi-cpufreq*, *intel_pstate*, *pcc-cpufreq*, etc) that implements the Advanced Configuration and Power Interface (ACPI) specification[39]. According to this specification, different pairs of frequency-voltage are defined and applied to the chip processing

[1] and device driver
[2] OpenACC is available in commercial compilers with incipient support is other commonly used compilers.

units. These pairs are known as Processor Performance States (*P-states*) and range from P_0 to P_n, with the higher index corresponding to a lower power consumption.

DVFS can be automatically applied by the operating system using power governing policies or explicitly defined using command-line tools. Power governing policies (governors) are generic to most of the drivers except for the *intel_pstate* that provides its own governors. In general, governors are based on simple models that perform frequency change decisions based on CPU load, CPU utilization and generic parametrizations[1]. For instance, the *ondemand* governor performs periodic checks on CPU-usage statistics and calculates a new frequency with a linear function based on the usage of the last period. Automatic power management is also applied in GPUs by adjusting clock frequencies depending on device load. DVFS can also be explicitly performed using manufacturers tools.

Some of these drivers and tools provide interfaces to specify frequency per core in multi-core systems. However, specifying voltage and frequency per core arises severe hardware architectural complexities, resulting in the unclear behaviour of the chip. The support and information provided by the manufactures to this feature are also unclear. The discussion of these architectural complexities is out the scope of this work, therefore, for simplicity, the discussion in terms of power consumption in this work is always in regard to the **full processing chip**. All the potential changes in frequency are applied equally to all processing components of the chip.

2.4 Addressing the Challenges

The available programming models provide the basic required tools to develop applications for current market devices. Most of them are tailored to a single associated device architecture and designed to be as flexible and as general purpose as possible. However, when combining multiple available devices together, multiple challenges emerge that raw standard programming models do not address.

Most common scientific applications are data-parallel where the workload is essentially proportional to the input data. They also generally resort to some type of iterative methods where the application is defined in multiple iterations over data and/or computed data with data dependencies within and between iterations (see Infobox *Applications and Data-dependencies*). Typically, these applications are developed following static and uniform distributions of workload, where the input data is equally divided across CUs for parallel processing. In distributed memory systems (Tier-4), nodes equipped with more recent and sophisticated CPUs and/or a GPU will potentially deliver far more performance, finishing the assigned work much faster than a neighbour node with older processors and no co-processors. In the presence of data dependencies, faster nodes will thus have to wait on slower nodes in order the get newly

[1] https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

computed data and continue with the computation. This will result in **node idle times and subsequently resource underutilization and poor scalability**.

> **Applications and Data dependencies**
>
> Data dependencies are arguably the most challenging aspect of parallel computing. With impact in every tier of parallelism, from ILP to supercomputers scalability, they not only dictate application performance but work scheduling strategies as well. The level of data dependencies of an application ranges from **embarrassingly parallel work**, where there are basically no dependencies between data items, to **tightly-coupled parallel work**, where all the work units may require information from any other work units at any time. An application with data dependency characteristics in between these two, can be classified as **loosely-coupled application**. In a data-parallel task-scheduling strategy, embarrassingly parallel work units can be easily submitted to a queue and dequeued for processing in any arbitrary out-of-order fashion (e.g. image pixels in a pathtracer engine). However, in a tightly-coupled application, a queueing approach may be unfeasible or inefficient and the scheduling strategy must account for application data dependencies resulting in completely different approaches (e.g. some parallel CFD simulations are tightly-coupled and typically resort to shadow or halo layers between processor boundaries to elide dependencies, impacting partitioning decisions and scheduling design).

The **performance imbalance** issue may also arise among devices (Tier-3). Different devices exhibit different performances depending on hardware architecture combined with application characteristics and implementation. Code divergence, memory access patterns, communication-computation ratio, are some of the features that define the workflow of an application which will impact device performance depending on the number of cores, cache models, execution model, etc. For instance, in a data-parallel task-scheduling approach, the task granularity becomes of crucial importance leading to tricky trade-offs that will define overall performance. Tiny tasks will increase parallelism and device throughput but dealing with a large number of tasks will incur in overheads from task creation and scheduling, increased communication and synchronization costs. Large tasks will counter these overheads, but will significantly reduce the degrees of freedom of a scheduling algorithm resulting in devices waiting for each other due to performances differences.

> **Data-parallel task-scheduling vs Functional task-scheduling**
>
> A data-parallel task-scheduling approach divides the computation into multiple tasks that perform the same computation to different data. Each task corresponds to a set of data elements e.g. a block of a block matrix multiplication, that is concurrently assigned to compute resources. In a Functional task-scheduling approach, tasks correspond to computational functions or kernels applied to the same or different data e.g. a pipelined execution – reading a matrix can be executed in parallel with the processing of a previous matrix.

Moreover, the performance imbalance issue is further aggravated in the presence of **dynamic workloads** typically present in numerical computer simulations. Having different performances across CUs becomes harder to address since the workload that needs to be properly scheduled is now dynamic and unpredictable. A static strategy that distributes the work across resources quickly becomes obsolete after a few iterations due to new work generated at runtime leading to huge performance losses.

Tackling the performance imbalance issue requires informed workload decomposition and re-distribution

mechanisms that existing APIs and runtime systems do not provide. The decomposition process is essential in a parallel computing system but finding the ideal sub-problem size is a challenging task. This requires a mechanism able to accurately model CU performances and provide that information to the partitioning system in order to devise a balanced workload partition. Estimating and measuring is influenced by several details that are particular to each CU but also related to application operations and behaviour. Such performance modelling mechanism must also be as less intrusive as possible in order to minimize measurement overhead and cluttering. In addition to the performance model, DLB mechanisms are required in order to redistribute the workload at runtime. These must perform **adaptive workload migration decisions** considering system load imbalances and devise a new balanced computational effort. Redistribution of work units potentially requires migrating complex data-structures across distributed and/or disjoint memory, requiring new data-migration routines and subsequent communication overhead minimization.

The parallelization and optimization approach of an application is typically associated with a specific architecture. However, given the plurality in terms of execution models in modern computing devices, programmers need to re-think their approaches when using different devices. For instance, an application that was designed and optimized for the CPU, will potentially deliver far less performance when executed in the GPU and vice-versa. With dynamic workload applications, **performance non-portability** also becomes a more compromising factor in maximizing efficiency. Massively parallel devices, designed for well structured and homogeneous work, will be severely affected by divergent code paths and scattered memory accesses generated by dynamic workload. This performance portability issue is not accounted for in standard programming models that are designed to express the execution model associated with a particular device or computing infrastructure. Furthermore, maintaining multiple implementations and developing new ones based on either architectural development and/or application requirements is highly counter-productive.

Unified execution models can be proposed that comprise the details of multiple architectures. Combined with an expressive and suitable API, this approach can provide a unified view of all the computing units and be complemented with dynamic workload scheduling while hiding the complex and diverse nuances of each device. It can be seen as a generic and automatic optimization tool that will increase productivity and potentially increase the performance extracted.

Communications between CUs play a crucial role in performance due to their **disjoint memory address spaces**. Transferring data between nodes in a network is one of the major bottlenecks in scalability. Similarly, co-processors are typically designed with their own memory, physically separated from main system memory. Application data must travel through a limited bandwidth bus (PCI-Express), which results in a potential performance bottleneck. In several devices, the available developing tools shift most of the data handling to the programmers. Data transfers must be explicitly managed and minimized for consistency and efficiency purposes. Commonly, the parallel approach and/or the algorithm must be reconsidered in

order to minimize data transfers, avoiding synchronization points or to mitigate these bottlenecks with other operations. These tasks can be delegated to a **data management system** that will perform automatic data transfers while minimizing communications by exploring data locality. The system can also try to overlap communication and computation that will mitigate communication overhead. Using an API, the programmer can register the data and the system will be responsible for all the management which will significantly increase developing productivity. In the case of devices where data transfers are automatic (e.g. GPUs with Pascal architecture and above) these challenges are partially addressed, however, delegating memory transfers to the driver results in loosing control of which and when data transfers occur. This inhibits optimizations such as data pre-fetching and computation overlap.

GPUs and other co-processing devices are deployed as co-processing boards and are typically used with libraries and programming tools developed by each of the different manufacturers. Despite the efforts of these manufacturers to use common languages such as C or C++ and standard specifications such as OpenCL, an application implementation **code is not portable**. This is due to several reasons, the most obvious one is the differences between execution models and architectures of each of the devices that are reflected in the programming models. To best express the features of their devices, manufacturers added specific primitives in the programming models and development tools, resulting in non-portable code.To address this issue, a **device-agnostic programming model** can be proposed that will hide specific primitives inherent to each device. It may work as wrapper offering a unified API to the programmer. This API may also provide access to all the features discussed above. The goal is to increase productivity allowing the programmer to focus on developing the problem.

Finally, the **power consumption** of large scale systems is converging to critical levels of impact and sustainability. In fact, reducing power requirements has been marked as one of the major goals for the forthcoming exascale era, with power efficiency having more focus when designing and optimizing HPC solutions. Addressing this challenge is not exclusively related to power itself, it **also requires considering the performance impact** since both are correlated. This challenge, however, is also aggravated by the heterogeneous nature of HPC systems and dynamic workload applications. Devising a strategy to reduce power consumption becomes non-trivial when facing a plurality of devices each with different power requirements, power usages, performances, tools, etc. It becomes even harder if the workload, which requires power for processing, is unpredictable. Different power consumption per CU, different performances per CU and an arbitrary workload results in an extremely complex decision process with multiple trade-offs that current out of the box power management systems do not address.

Power management in these conditions can be achieve by proposing a power model that will estimate the power consumption of each CU. This will **raise awareness for the different devices** in the system but will also enable runtime predictions of power consumption with dynamic workloads. This information can then be combined with a performance model in order to estimate performance impact. The resulting model can be used in a decision or optimization process that will devise runtime power assignment

decisions towards minimization of power consumption and maximum performance.

Summarizing, performance and power modelling, DLB, data management systems, unified programming and execution models, among others, are some of the techniques that are required for programmers to face the challenges posed by Heterogeneous Systems (HS) and dynamic workload applications. These techniques are not provided by the set of standard tools used by the developers and this thesis hypothesizes their use in order to increase productivity, performance, scalability and, ultimately, cost-effectiveness. Some approaches to these techniques have been proposed and evaluated in literature and will be individually discussed in detail in the following chapters along with the proposed approaches by this thesis.

Chapter

3 Heterogeneous Single-node Systems

Contents

This chapter discusses an approach to address the challenges emerged from single-node heterogeneous parallel systems. It proposes a runtime system composed of a programming and execution model, together with workload scheduling mechanisms and data management tailored for irregular applications. The runtime system is evaluated with multiple compute resource configurations as well as different regular and irregular workloads.

3.1 Introduction

In this chapter the particular challenges exhibited within single node multi-device systems (Tier-3 and below) are addressed. These systems are composed by multiple devices, including multi-core CPUs — that also act as the host device for the node – along Intel Xeon PHI's, GPUs, DSPs, FPGAs, that are usually packaged as co-processing boards. Although heterogeneity is now ubiquitous, some challenges emerge from this plurality of devices and, in particular to HSNS, from the architectural differences and execution models. In order to fully leverage the whole system, addressing these challenges is of crucial importance.

As discussed in Section 2.4, co-processors have **disjoint address spaces** between themselves and the host CPU, usually interconnected by PCI-Express bus which is a potential performance bottleneck. In some devices, data transfers must be explicitly coded, while in others (e.g. Pascal GPUs and above) data is automatically transferred. In both scenarios, data transfers must be managed and minimized for optimization and efficiency purposes. The different architectures typically exhibit different execution and programming models and are made available with different development tools, severely impacting on both code and performance portability. Applications are also designed and optimized to fully utilize each device computing capability according to the device specific architecture and execution model, reducing development productivity. Moreover, the application's workload has to be distributed and balanced among the multiple devices (Tier-3), and, within each device, among its multiple processing units (Tier-2). Addressing these issues requires the development and adoption of Tier-3 and Tier-2 scheduling mechanisms towards maximum performance extraction.

These challenges are aggravated if the target applications exhibit a dynamic behaviour. In this contribution a particular set of dynamic applications are targeted which exhibit certain characteristics that particularly hinder the performance of HSNS. These applications are defined as **irregular applications** and are characterized by irregular data structures, irregular control flow and/or irregular communication patterns [40]. These particular workflows cause load imbalance, code divergence and uncoalesced memory accesses, all potentially resulting on significant performance losses in HSNS. They particularly hamper the performance in wide SIMT devices, such as GPUs. The hardware work dispatch units within these devices are optimized for homogeneous regular workloads, maintaining high utilization of SIMT lanes

and thus exhibiting remarkable performance improvements over CPUs for regular applications. Irregular applications, however, have the potential to follow different code paths and perform scattered memory accesses within the same lane (See Section 2.1.3), resulting on code divergence, increased memory access latencies and resource underutilization. In order to fully exploit these devices, maximum levels of occupancy should be guaranteed.

Irregular applications constitute an **important class of algorithms** that are present in well-known scientific applications, such as n-body simulations, data mining, decisions problems, optimization theory, pattern recognition and meshing among others [10], [40]. A particularly relevant subset of irregular applications are Monte Carlo simulations [14], widely used in many knowledge areas, such as financial engineering and valuation [41], [42] or physically based simulation of light transport within complex media [43], [44], among many others. Monte Carlo simulations perform multiple Markov random walks within the domain and then average the results of such random walks in order to obtain an estimate of the metric of interest. Since both the direction and the length of the random walk are stochastically generated, this results on an irregular workload, exhibiting load imbalances, control flow divergence and irregular memory accesses.

A framework is proposed that specifically addresses **development and execution of data parallel irregular applications in heterogeneous single node systems** towards increasing its efficient utilization while maintaining high programming productivity. The framework is essentially composed by a unified task-based programming and execution model for data parallel irregular applications, together with high-level programming abstractions and scheduling mechanisms that transparently partitions the data domain into tasks and deals with all Tier-3 and Tier-2 workload distribution and balancing. The Tier-2 scheduling resorts to **persistent kernels** and a queuing system that will also orchestrate the work leveraging Tier-1 parallelism (SIMD). A data management strategy is also proposed that transparently guarantees that required data is readily available on each task's addressable memory space. These components and their integration in a framework constitute part of the hypothesis of this thesis towards the efficient harnessing of the combined challenges posed by Tier-3, Tier-2 and Tier-1 systems and dynamic irregular workloads.

The main contributions are thus the **unified execution and programming model and the integration of persistent kernels** on the proposed framework as the solution to handle irregular workloads. An implementation of the framework is presented, together with an experimental assessment of its ability to efficiently handle regular and irregular workloads and a comparison with a state-of-the-art competitive framework. Validation of the above hypothesis is performed in CPU+GPU heterogeneous platforms and with emphasis on scheduling irregular workloads within the GPUs. Four case studies are used: a regular matrix multiplication, an irregular n-body problem using the Barnes-Hut algorithm, an irregular path tracing based renderer and an irregular simulation of light transport with fluorescence within multi-layered tissues.

3.2 Related Work

Several programming models and frameworks have been proposed that aim at hiding some of the challenges posed by HS in order to increase development productivity. **HMPP** [45] is one of the first CPU+GPU programming models aiming at handling devices and use them without the need to re-write the applications. The model introduces per-device *Codelets* as a means to express the application functionality, along with primitives for execution and data transfers. However, it lacks a runtime system and scheduling policies that hide some of the remaining challenges such as load balancing.

Harmony [46] proposes several techniques to address HS challenges and approach the associated complexity. The work assesses and validates some solutions presenting results of a unified execution model, control decisions and a shared address space. **Merge** [47], is focused on portability issues providing a compiler and runtime system and following a map-reduce approach for scheduling. The authors claim that Merge is applicable to different HS and applications are easily extensible and can easily target new architectures. These approaches are focused on the challenges that the plurality of architectures pose, such as code portability and productivity. However, they do not properly address data management, scheduling and load balancing.

XKappi [48], **Legion** [49], **Qilin** [50], **MDR** [51] and **StarPU** [52] are frameworks that provide high-level programming abstractions for multi-device systems, integrated data management and enhanced scheduling mechanisms. Both XKappi and Legion target multi-device executions with focus on data parallel scheduling. Techniques such as locality aware work stealing and task-dependency Direct-acyclic Graph (DAG) scheduling are explored coupled with a suitable programming model. In addition, Legion provides a more sophisticated support for irregular data structures accounting for applications such as graphs processing. Qilin provides enhanced compiling features and a performance modelling mechanism while MDR focuses on scheduling, proposing a scheduling approach entirely based on online history-based performance modelling together with an analytical model for communications.

StarPU has more advanced data-management and sophisticated scheduling techniques. It provides a unified execution model combined with a virtual shared memory and a performance model working together with dynamic scheduling policies. The runtime also provides several data-management features: automatic work decomposition and data transfers, communication and computation overlapping, data pre-fetching and data locality aware scheduling, among others. The scheduling resorts to an heterogeneous tailored algorithm known as the **Heterogeneous Earliest Finish Time (HEFT)** [53]. The data management system used in this contribution is strongly inspired on that of StarPU; it uses the same cache protocol with lazy consistency and keeps the programmer agnostic to data movements.

Some of the challenges of HS have been preliminary addressed in [54], where a framework is proposed that provides an unified programming and execution model combined with a data management system.

The contributions proposed in this chapter use some of the developments described in this work, combined with mechanisms to efficiently execute irregular applications.

These frameworks address some of the challenges associated with HS, however, they do not tackle the specific issues associated with irregular applications. Tier-3 scheduling and work decomposition are based on previously sampled information where the performance of a small subset of work is generalized for the whole domain – irregular applications are particularly sensitive to these generalizations, since the workload varies among data elements in an unpredictable manner. Tier-2 scheduling is also not properly considered. Irregular data parallel workloads require performing some fundamental operation to each data element an unknown number of times; e.g., on a pathtracer the length of the path per pixel, i.e., the number of rays, is unknown and varies unpredictably across screen space – path tracing can thus be seen as tracing a previously unknown number of rays. In the GPUs for instance, this irregularity would lead to code divergence and huge resource underutilization.

Some approaches have been proposed in literature that transparently map irregular applications to wide SIMT devices, balancing the workload across the device CU and alleviating the programmer from the need to explicitly deal with this issue. Cederman et al. [55] evaluates the use of dynamic load balancing methods based on queues with lock-free and work-stealing mechanisms within the GPU. Tzeng et al. [56], inspired by the proposals of Aila and Laine [57], introduced a task management system based on persistent kernels and queues, which maximizes CUs utilization and load balance. Persistent kernels produce and consume work using a queuing system, avoiding the multi-pass approach and allowing load redistribution through a task donation/stealing mechanism. Softshell [58] also proposes a three-tier scheduling model for the GPU that aims to replace the current built-in scheduling systems. It also works on top of a persistent kernel approach similar to Tzeng's, proposing an aggregation scheme of threads and work items, sorting work items by priority and using queues to manage work items.

The Tier-2 scheduling approach in this contribution is inspired by Tzeng's task management system, integrated in the proposed framework that provides transparent access to the task system through the proposed programming model and API.

3.3 Proposed Approach

In order to address the discussed challenges posed by the HS, a framework is proposed that encompasses multiple features that work together at runtime. The aim of the framework is to increase productivity whilst transparently improve performance by increasing resource utilisation. This section provides a detailed description of the proposed programming and executions models, programming interface and system architecture that compose the framework, in particular how they tackle the challenges posed when efficiently exploiting heterogeneous systems with irregular applications.

3.3.1 Programming and Execution Model

The proposed framework uses a host-device system model, with applications being composed by a **host control program** (HCP) plus one or more **computation kernels** and respective **data sets** (Figure 3.1). The HCP runs on the CPU and is responsible for data registration and partitioning, synchronisation and enforcement of dependency constraints among computing kernels. Kernels express the application functionality and are executed on the system devices (including the multi-core CPU). They apply some computation to all elements of a data set; in this sense, kernels express data parallel problems and the application of a kernel to one data element is referred to as a **basic work unit**. Basic Work Units (BWUs) within the same job are assumed to exhibit no data dependencies among them. It is the programmer's responsibility to provide implementations of the kernels for each device architecture.

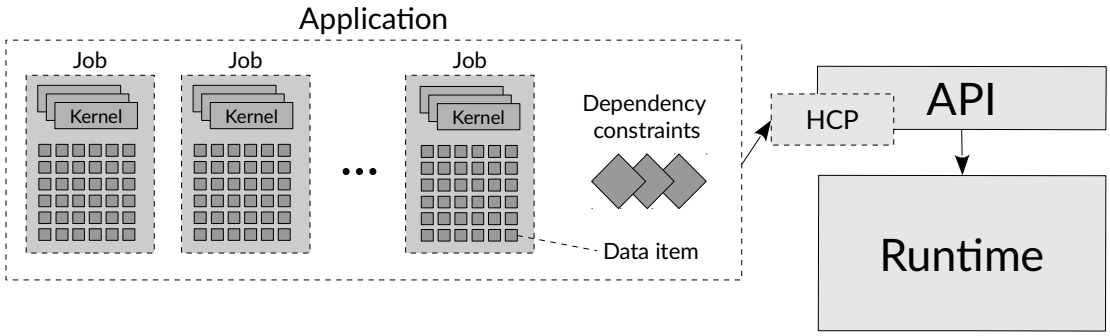


Figure 3.1: Application specification and HCP components. Application jobs and dependency constraints are submitted to the system by implementing the HCP using the API

An application consists on one or more **jobs**, each consisting on applying a computation kernel to a data set. The runtime system partitions the job data set into blocks of BWUs, referred to as **tasks**, whose execution is dispatched onto available devices. The data set partitioning and dispatching is transparent to the application programmer. **Dependency constraints** among jobs must be explicitly specified by the HCP using system primitives, otherwise they may execute concurrently. Tasks are executed out-of-order and completely transparent to the application programmer. Partitioning is, however, dependent on application specific data representation; the programmer is thus required to implement a provided interface for a callback method that will of create arbitrarily sized data partitions upon system demand; this method, renders the runtime system independent on data representation.

Data domains are used as a mechanism to transparently manage data. These, inspired by Partitioned Global Address Space based languages such as Chapel [59], encapsulate all the information required for the system to manage user data, including data location and transfers. **Hierarchic data partitioning** is internally supported by a hierarchy of sub-domains, which represent smaller regions of the data set. The runtime system converts domain global indexes to task local sub-domain indexes, thus transparently supporting arbitrarily sized tasks; only the notion of domain is exposed to the programmer. The data

management system uses a **MSI**[1] **cache coherence protocol**, similar to StarPU [52], to enable data replication and ensure consistency among replicas, which combined with lazy data transfers reduces data movement overheads. Data pre-fetching and overlapping of asynchronous data transfers with computation are also supported to further reduce communication overheads[2]. The runtime system does not ensure data consistency among concurrent jobs, i.e., if different jobs update the same data, they must be explicitly serialized by the HCP using system primitives.

In order to effectively handle both regular and irregular workloads the runtime system supports two types of kernels: **consumer** and **consumer-producer** kernels. The choice of the type of the kernel depends on the application and will define the internal execution model of the runtime. Consumer kernels are associated with regular applications and imply the complete processing of a data element. In regular workloads the imbalances among BWUs within the same task are unlikely and thus there is no need for further complexity. Consumer-producer kernels are used within a persistent kernel and are targeted for irregular workloads by addressing the highly unbalanced computational and memory demands across data elements. On wide SIMD/SIMT architectures this would result on increased utilization of the devices' CUs.

3.3.2 Consumer vs Consumer-producer Kernels

Irregular data parallel workloads require performing some fundamental operation to each data element an unknown number of times; e.g., on a pathtracer the length of the path per pixel, i.e., the number of rays, is unknown and varies unpredictably across screen space – path tracing can thus be seen as tracing a previously unknown number of rays. The consumer-producer kernel will basically define the BWU using this sub-operation rather than the complete processing of a data element. This is basically the main difference between the two types of kernels: the consumer kernel processes the whole data element while the consumer-producer kernel fragments this operation into multiple identical ones. A consumer-producer kernel applies this BWU to a data element and, if required by the algorithm, dynamically generates a number of new BWUs, which are then rescheduled within the device by resorting to a queuing system. This approach is used to address the Tier-2 scheduling challenge and allows balancing the irregular workload and increasing resource utilization within each device.

In the pathtracer example, a consumer kernel would follow the entire path, eventually leading to imbalances when paths have different lengths; a consumer-producer kernel would follow a single segment of the path, i.e., a ray (and, eventually, associated shadow rays), generating a new BWU (a new path segment) at each intersection point until the path finishes. By rescheduling the newer generations of BWUs

[1] Modified-Shared-Invalid

[2] Note that, even though some devices support transparent data transfers, only by having control of what and when is transferred allows for these optimizations

within the device, imbalances at Tier-2 level due to the irregularity of the workload can be minimized. It is the responsibility of the application programmer to decide whether a consumer or a consumer-producer kernel is to be used for each job.

Note that consumer kernels are launched by the runtime system which transfers all the required data automatically to the device. This means that the consumer kernel **allows the application programmer to freely map the task workload onto the device resources**. This also grants him complete control over the device and enables the use of lower level programming tools, such as CUDA [24], or highly optimized libraries, such as CuBLAS [60] or the Intel Math Kernel Library [61]. Consumer-producer kernels, on the other hand, are under control of a running a persistent kernel [56], which calls the consumer-producer kernel, provided by the application programmer, in order to process BWUs – thus precluding the utilization of such third party libraries.

A final crucial feature in consumer-producer kernels is the scheduling of **work in batches of SIMD width length**. Since consumer-producer kernels are target for wide SIMD/SIMT architectures, BWUs are automatically grouped in sets of *simd-width* length. The benefit of this grouping is, on one hand, to match the execution model of the GPU, for instance, where *simd-width* instances of threads are simultaneously scheduled and executed in lock-step (warp), and, on the other hand, to promote coherent execution from which SIMT devices will leverage. For instance, in the path tracing example and in a NVIDIA GPU, the consumer producer kernel groups 32 path segments that have been extracted from a task and simultaneously executes them. Since neighbour primary rays of a pathtracer will hit neighbour geometry, the GPU is able to benefit from data coherence and increased performance is achieved.

3.3.3 Programming Interface

The HCP is the entry point to the framework and it is where the programmers specify all the data and functional requirements of the application. The API definition leverages the object oriented inheritance paradigm providing higher flexibility to the programmer when expressing their applications. Code 3.1 illustrates a simplified example of a HCP for the pathtracer application. Domains for the resulting pixels radiance and for the geometry are created and linked to the corresponding user data structures (Lines 6 and 7). A job is then created, domains are associated and device kernels are specified (Lines 9 to 15). The job is then added for execution and the HCP is instructed to wait for the job to finish (Lines 17 to 18). Finally, the computation results, stored onto a domain, are gathered to host memory (Lines 19). Note that, apart from associating the kernels to devices, the HCP is agnostic to any computational resource details as well as any work partitioning and scheduling policies.

Pathtracer host control program

```
1  HCP_PATHTRACER() {
2      RGB* pixelsRadiance = new RGB[PIXEL_COUNT];
3      Geometry *geometry = new Geometry();
4      (...)
5
6      Domain<RGB>* d_pixelsRadiance = new Domain<RGB> ("RAD", pixelsRadiance,
           dim_space(0, PIXEL_COUNT));
7      Domain<char>* d_geometry = new Domain<byte> ("GEO", geometry, dim_space(0,
           GEOMETRY_SIZE));
8
9      Job_PATHTRACER* t = new Job_PATHTRACER();
10     t->associate_domain(d_pixelsRadiance,d_geometry,...);
11     t->camera = CAMERA;
12     t->SPP = SPP;
13     (...)
14     t->associate_kernel(CPU, &CPU_pathtracer_kernel);
15     t->associate_kernel(GPU, &GPU_pathtracer_cpkernel);
16
17     AddJob(t);
18     WaitForAllTasks();
19     GetDomain(d_pixelsRadiance);
20 }
```

Code 3.1

Code 3.2 presents a high level excerpt of a consumer kernel for pathtracer on the GPU. Within the kernel, the appropriate domain is gathered from the runtime system, followed by gathering the appropriate BWU – on this example this is represented by the first ray the kernel will have to trace and shade (Lines 3 and 7). Then the iterative intersect and shade of the sample path is performed, using Russian roulette to stochastically decide whether the path should continue or not. The result of this BWU is then written onto the domain. Code 3.3 illustrates a consumer-producer kernel for the same application and device. The main difference is the loop removal since the processing of a sample is now transformed into a sequence of an unknown number of basic units. After intersection and shading, and depending on the result of the Russian roulette (Line 11), a new BWU is created and submitted to the runtime system for scheduling within the device (Line 12). This new BWU will be computed by the same consumer-producer kernel and the required data (e.g. pixel id) is inherited from the current task. Finally, the result is accumulated onto the domain.

Pathtracer GPU **consumer** kernel

```
GPU_pathtracer_ckernel(TASK* task) {
   Domain<RGB> pixelsRadiance;
   task->GetDomain("RAD", pixelsRadiance);

   RayHit hit;
   RGB result_rad;
   Ray ray = getRay(task);

   do {
      Intersect(ray, hit, ...);
   } while (ShadeAndRussianRoullete(result_rad,...));

   int pixel_id = getPixelID(task);
   pixelsRadiance->at(pixel_id) = result_rad;
}
```

Code 3.2

Pathtracer GPU **consumer-producer** kernel

```
GPU_pathtracer_cpkernel(TASK* task) {
   Domain<RGB> pixelsRadiance;
   task->GetDomain("RAD", pixelsRadiance);

   RayHit hit;
   RGB result_rad;
   Ray ray = getRay(task);

   Intersect(ray, hit, ...);

   if (ShadeAndRussianRoullete(result_rad,...))
      newBWU();

   int pixel_id = getPixelID(task);
   pixelsRadiance->at(pixel_id) += result_rad;
}
```

Code 3.3

Together with the data partitioning method and additional kernels for each supported device architecture, these code blocks illustrate all the functionality the application programmer has to provide in order to benefit from multi-device data management and dynamic workload distribution and balancing. To further increase transparency, a generic specification of kernels can be provided that would support different architectures therefore further reducing the user provided code and programming effort.

3.3.4 System Architecture

Figure 3.2 illustrates the runtime system architecture and how the different modules cooperate with each other. All the communication between the application and the framework is done through the API, which is one of the main entities along with the Scheduler, Performance Model (discussed in Section 3.4.1) and Data Management System. The system has a central job queue from where the Scheduler dequeues jobs upon device request and, using the data partitioning methods and the information provided by the Performance model, produces the proper sized task and assigns it to the device. Each device in the system has its own queue and associated control thread running on the host, enabling asynchronous data and control flow using system messages. This distributed-like system increases scalability since the devices will request work asynchronously and process tasks' data concurrently. The devices' queues support an execution window of tasks enabling computation overlapping with data transfers and data pre-fetching. The kernel that runs on the device can be of two types depending on the type of the application as discussed in Section 3.4

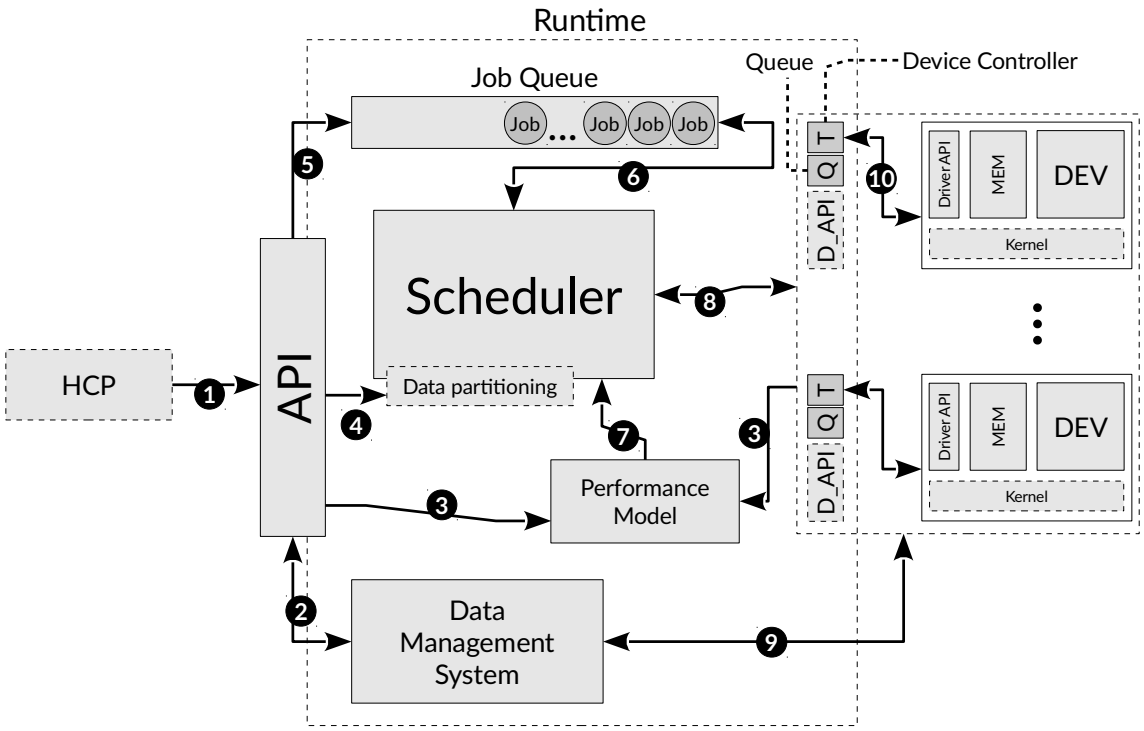


Figure 3.2: Runtime architecture and workflow.

The detailed description of the workflow in Figure 3.2 is as follows: (1) The HPC provided by the programmer uses the runtime API to define and submit application jobs and dependency constraints; (2) using the API, the user will register and gather data to the Data Management System (DMS); (3) the number of data elements to process is provided as well as user-provided information to the performance model if applicable; task execution information is also provided from the devices to the Performance Model; (4) application-specific data partition methods defined; (5) jobs are enqueued in the main queue; (6) the scheduler dequeues and enqueues jobs or tasks from the main queue; (7) the scheduler assigns a job to a device, reasoning about job workload and device compute capabilities which will potentially trigger data partitioning methods to create a properly sized task; (8) task is enqueued in device queue; (9) the device controller signals data movements required for the task; (10) the device controller signals for task execution using the user-provided kernel.

Each device architecture supported by the framework requires the development of a Device API (D_API) implementation, allowing the framework to perform low level operations such as initiating computations or copying data to/from the device. The Device API is transparent to application programmers, but explicitly managed by the framework developers. For instance, a system with three NVIDIA GPUs and two CPUs requires two Device API implementations, one for the GPUs and another for the CPUs. For each of these implementations, alternative programming and execution environments might be selected; for example, either CUDA or OpenCL might be used to control the GPUs.

3.4 Workload Scheduling

One of the major goals of this contribution is to devise and evaluate scheduling mechanisms that allow for increased performance in Tier-3 and Tier-2 parallel systems. In other words, the goal is to minimize execution time and, in this chapter, the proposed approach to achieve this is to increase resource utilization by keeping the workload distribution well balanced among available computational resources. **Tier-3 scheduling** consists in partitioning the job's workload into tasks and assigns them to individual devices, whereas **Tier-2 scheduling** distributes a given task BWUs among the device internal computational units. In the proposed framework, each of the two parallel tiers is addressed with different mechanisms. Even though scheduling is a major component of the proposed framework it is transparent to the application programmer.

3.4.1 Tier-3 Scheduling

Tier-3 scheduling is performed by resorting to a demand driven strategy, where tasks are assigned upon device request. When a device finishes a task it signals the scheduler, indicating that it is available for further processing. The scheduler then fetches a job, decides the new task size by applying the

partitioning strategy described below, applies the data partitioning method to get the proper sized task and submits the task for execution to the requesting device.

Demand driven has been preferred over the HEFT scheduling algorithm [53], which is used by StarPU, since the latter makes its decisions based on an initial sampling of the workload behaviour. However, the behaviour of irregular workloads is mostly unpredictable by definition and thus the authors of this work conjecture that HEFT is not appropriate for this kind of workloads. The demand driven behaviour is more suitable approach to cope with a wide range of workload profiles and with devices with diverse computing power. By partitioning a job's workload into a number of tasks larger than the number of devices and then assigning tasks on demand it adapts to both the workload requirements and the devices' capabilities.

However, scheduling overheads are also dependent on the number of tasks. A heterogeneous system is expected to have devices with very different computing powers, which would require a large number of tasks in order to maintain load balance, severely impacting on scheduling overheads. The total number of tasks can be reduced by tailoring the task size to the relative computing power of the device where it is being scheduled; this is the responsibility of the work partitioning strategy.

Let C_d represent the computing capability of device d, defined according to some performance model. Results presented on Section 3.6 are based on the devices' theoretical peak performances, as announced by the respective manufacturers. This might not be the metric that guarantees the best results, particularly for irregular workloads. However, it is beyond the scope of this work to select and evaluate the most appropriate performance modelling technique. In fact, the proposed framework takes a modular approach towards the performance model, allowing it to be replaced without impacting on the remaining runtime system architecture. This modularity assures that more efficient performance models and appropriate metrics, eventually resorting to dynamic approaches, can be used in the future. C_d is normalized according to Equation 3.1 to represent relative computing capability with respect to the other devices present on the heterogeneous system. $T_{devices}$ is the total number of computing devices.

$$\bar{C}_d = \frac{C_d}{\max\left(C_1, ..., C_{T_{devices}}\right)} \tag{3.1}$$

The size of the task to assign to the requesting device, expressed in terms of the number of data elements to process (or BWUs), is then given by Equation 3.2

$$\frac{N}{dd} \times \bar{C}_d \tag{3.2}$$

where N is the job's total number of BWUs and dd is a system constant that allows control over the tasks' granularity, assuring that the total number of tasks is significantly larger than the number of devices, as required for a demand driven strategy to be able to properly balance the workload.

For instance, consider a system composed by a GPU and a CPU where the performance model dictates

that the normalized relative compute capabilities are 1.0 and 0.3 to the GPU and CPU, respectively, and let *dd* be equal to 10. Upon receiving a work request, the scheduler will fetch a job, say with 1000 BWUs, and assign a task with 100 BWUs if the requesting device is a GPU or with 30 BWUs if it is a CPU.

3.4.2 Tier-2 and Tier-1 Scheduling

Tier-2 and Tier-1 scheduling are targeted to make use of the **consumer-producer kernels** and applies only to irregular workloads; for regular workloads, the programmer can use consumer kernels having complete control over the device as described on Section 3.3.2. Tier-2 scheduling exploits the fact that irregular workloads can be seen as applying some fundamental operation to each data element an unknown and unpredictable number of times; the BWU is thus redefined as this fundamental operation, rather than the complete processing of a data element. This view enables a work-spawn strategy where the execution of a BWU leads to the potential spawning of one or more dynamically generated new BWUs. In order to efficiently handle this mechanism within a SIMD/SIMT device, a generic pipeline is implemented that features most of the techniques proposed by Tzeng et. al [56].

A GPU is a SIMT device that schedules bundles of threads with the same cardinality as a SIMD lane – on most NVIDIA GPUs these bundles contain 32 threads and are referred to as warps. Since warps are executed in lockstep, code divergence and uncoalesced memory accesses should be minimized for performance maximization. However, irregular applications tend to exhibit divergence and unpredictable memory accesses. In order to address these issues, the consumer kernel is replaced with built-in persistent kernel implementation. Note that this means that the runtime system manages the kernel execution within the device, whereas with consumer kernels the application code has complete control of kernel execution within the device.

The execution model of the persistent kernel follows a **SIMD lane programming approach** that cooperates with the hardware scheduler to manage these lanes. As illustrated in Figure 3.3, each lane is endued with two local queues for getting work to consume and to store locally generated new BWUs, respectively Local Inbox Queue (LIQ) and Local Outbox Queue (LOQ). Work is shared among different SIMD lanes by using a device Global Inbox Queue (GIQ) with a try-lock mechanism to avoid contention. This fetch of work from a shared queue enables Tier-2 scheduling since each lane will be computed by a different stream multiprocessor. Each lane will fetch a bundle of 32 BWUs (on NVIDIA GPUs) and call the user-provided consumer-producer kernel, using a callback mechanism, in order to process all fetched BWUs . Dynamically generated BWUs are stored on the LOQ and eventually moved to the GIQ in order to allow execution on other SIMD lanes. This enables transparent access to SIMD lane programming and Tier-1 scheduling by the application programmer, which is now able to maximize application code convergence and coalesced memory accesses assuming that all 32 BWUs will be executed within a single lane.

The details of the workflow as illustrated in Figure 3.3 as follows: (A) if space available in local inbox

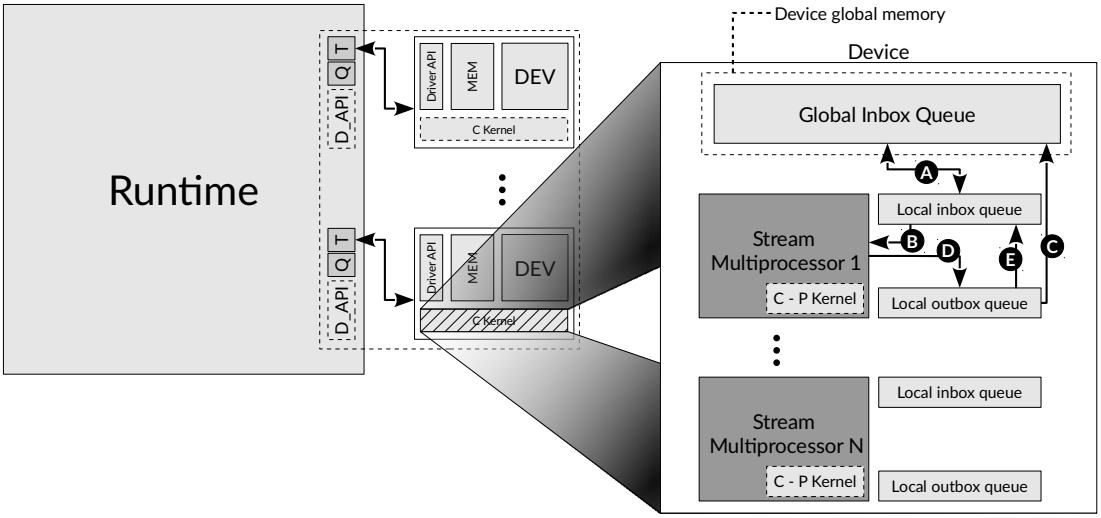


Figure 3.3: Persistent kernel architecture and workflow.

queue (LIQ) try-lock global inbox queue (GIQ) and dequeue tasks; (B) Retrieve tasks and execute them using the user-provided consumer-producer kernel; (C) If there is not enough room in local outbox queue (LOQ) and in LIQ to store all secondary tasks, force GIQ lock and enqueue all the elements from the LOQ; (D) Store generated tasks in LOQ; (E) Enqueue in LIQ elements from LOQ. If LIQ is full try-lock GIQ

3.5 Evaluation Approach

This section presents the evaluation approach and methodology of the proposed model and associated framework. It describes the applications used, namely a regular application – matrix multiplication (MM) – and three irregular applications – a Barnes-Hut n-body simulation (BH), a pathtracer (PT) and a Fluorescence simulation (FL). It also describes some of the metrics used and the computing system.

3.5.1 Applications

As a regular application, only the consumer kernel is provided for the **matrix multiplication**. In order to compute an element C_{ij} of the result matrix, the kernel performs a dot product between the row A_i and column B_j of the factor matrices. The kernel uses the CuBLAS [60] and the Intel Math Kernel Library [61] optimized libraries for the GPUs and CPUs kernels, respectively. A reference version executing on a single GPU was developed with CuBLAS for performance comparison purposes.

As an irregular application, the **Barnes-Hut (BH)** algorithm [17] casts an n-body simulation as an hierarchical problem, reducing its complexity to $O(N \log(N))$. The goal is to compute the force exerted on each particle of the data set by all other particles of the same set. The BH algorithm orders the particles by resorting to an octree (in 3 dimensions). When computing the resulting force, if a voxel is farther away

from the particle being processed than a given threshold, then all the particles contained in that voxel are approximated by their center of mass and the sub-tree associated with the voxel can be pruned. The unpredictability of which nodes of the octree will be visited for each particle renders the workload irregular.

A consumer kernel will, for each particle in the data set, traverse the octree, deciding which nodes to visit and which to prune and finally computing the resulting force – the basic work unit is thus computing the force for one particle of the data set. A consumer-producer kernel entails visiting one node of the octree and deciding which of its children to visit and which to approximate. All those children nodes that have to be visited result on the generation of new basic work units, which will be rescheduled within the device by the runtime system. On wide SIMD/SIMT devices, such as the GPUs, basic work units will be executed in groups with the same cardinality as the SIMD lane width (32 for current NVIDIA GPUs). In order to increase coherence within each SIMD lane, particles are initially sorted such that neighboring particles have high probability of being scheduled onto the same SIMD lane [12]; neighboring particles have high probability of visiting the same regions of the octree.

Monte Carlo Path Tracing (PT) is a well known ray tracing based rendering algorithm. It entails following light paths from the eye into the scene; at each intersection point radiant flux is gathered from the light sources using a given number of shadow rays and the continuation of the path is stochastically decided using Russian Roullette; if continued, a new ray is spawn, its direction being stochastically determined. The Russian Roullette path termination approach and the stochastic direction of each new ray render the workload irregular. On wide SIMD/SIMT architectures, coherent path tracing [13] is used, where the random numbers used to decide about path termination and new ray direction are the same for all threads within a SIMD lane. This will make paths within the same SIMD lane coherent (same length, same overall directions), which results on perceivable image artifacts; these artifacts are eliminated by shuffling the paths on the image plane before tracing them, thus avoiding spatial neighborhood among coherent paths [13].

A consumer kernel entails processing the whole path, whereas a consumer-producer kernel processes a segment of the path, i.e., one ray plus associated shadow rays and, if the path is continued, generates a new basic work unit with the new ray. The image plane is divided into multiple pixels and in order to increase image convergence multiple samples (i.e. light paths) are taken per pixel (SPP). Each sample is processed independently and the more samples, the better the image convergence, but the workload increases and more irregular paths are processed. The SPP parameter will be used to express the workload size as it is one of the parameters with major impact in image rendering and also impacts algorithm irregularity, which is addressed in this work. The basis pathtracing code was extended from the **SmallLux renderer** (recently re-branded as LuxCoreRender)[62]; a reference version of SmallLux running on a single GPU is used for performance comparison purposes.

The Monte Carlo simulation of light transport with **fluorescence** in multi-layered tissues (FL) is frequently

viewed as a reference method, whose results can be used to validate other less demanding methods [44]. It is based on following a packet of photons along random walk steps within a multi-layered media with complex structure, the size of each step being stochastically generated according to the media optical properties. After each step a fraction of the photon packet's energy is absorbed and a new step and scattering direction are stochastically chosen according to the current tissue layer properties. When a boundary between different layers, or between a tissue layer and the exterior, is crossed by the packet it might be either entirely transmitted into the new layer or reflected back into the same layer; this decision is once again made by resorting to a stochastic process and the optical properties of both layers. The random walk is continued until the photon packet exits the tissue or its termination is decided by Russian Roullette.

Fluorescence emission is simulated by deciding, after each step, whether a fraction of the absorbed energy, as given by the quantum yield optical property of the tissue layer, is re-emitted as a new fluorescent photon packet with a different wavelength; this decision is made by resorting to Russian Roullette. Fluorescent photon packets are propagated through the media using Monte Carlo simulation, with the same algorithm as the original excitation packets, except that they will not generate further fluorescent packets since their wavelength will not trigger this phenomenon. The basic work unit for a consumer kernel entails simulating all steps of a photon packet and respective fluorescent packets until they exit the media or are terminated by the Russian Roullette process. The consumer-producer kernel processes a single step of a photon packet random walk; a new basic work unit is created if the random walk is continued and an additional one is created for each emitted fluorescent packet.

Even though PT and FL resort to Monte Carlo simulations, the associated workloads exhibit some fundamental differences. The former entails tracing rays through the scene 3D volume, which is a computational expensive procedure, whereas the latter does not involve any tracing. In fact, FL just requires advancing the photon packet position along the random walk step direction; boundary crossing among layers is verified by checking the Z coordinate, since the modelled layers are aligned with the XY plane and thus all boundaries are perpendicular to the Z axis. Consequently, the basic work unit for the consumer producer kernel involves much less computation for FL than for PT. Additionally, for FL all photon packets are shot into the media through the same infinitesimal point, i.e., all random walks have the same origin. This is in contrast with the PT application where all paths initiate at different points of the image plane. This particularity hinders the application of coherence increasing techniques, such as the coherent path tracing technique used for PT. On wide SIMD/SIMT architectures, and for the consumer kernel, threads within a SIMD lane are thus expected to be more incoherent for FL than for PT, exhibiting larger code divergence, load imbalance and irregularity of memory accesses; the consumer producer kernel has the opportunity to minimize load imbalances within a device since new basic work units are rescheduled after each random walk step.

Furthermore, in FL a photon packet can contribute to any voxel within the grid embedded in the tissue,

whereas in PT a path only contributes to the pixel where it is originated: contention in memory writes, which are solved by resorting to atomic operations, is thus much more frequent in FL than in PT. Also, each task in PT requires a number of memory management operations, such as dynamic allocation and data copying, which is not required in FL. This is due to the fact that each task entails generating a tile of the image plane which is dynamically allocated by each device; such requirements do not exist in FL, where the above referred grid of voxels is only allocated once on each device, given that any thread can write to any voxel and the grid is much smaller than the finely sampled image plane. Such memory management operations represent an implementation penalty that might harm PT's efficiency. Finally, PT basic work units with the consumer-producer kernel have a branching factor of 1, i.e., after tracing a ray in the path if the random walk continues a single new task is generated with the new secondary ray. FL can have a branching factor of 2, since a new fluorescent photon packet can be created; the higher branching factor will impact on the results.

3.5.2 Heterogeneous Systems Metrics

Speedup, $S(p)$, and **efficiency**, $E(p)$, are two metrics often used to report and analyse the performance of homogeneous parallel systems with p processors. If T_p and T_1 are the execution times of the parallel and uniprocessor systems, respectively, then these are given by Equations 3.3 and 3.4. $S(p)$ is a measure of how faster the parallel system is than a sequential one and $E(p)$ constitutes a measure of resource utilization.

$$S(p) = \frac{T_1}{T_p} \tag{3.3}$$

$$E(p) = \frac{S(p)}{p} \tag{3.4}$$

The problem with the above metrics is that they are defined for the homogeneous case, where all p processors are identical. Similar metrics have been defined for the heterogeneous case [52], [63] and are used on this work to analyse the experimental results.

Let W define the workload associated with solving a given problem and T_{dev} be the execution time of that workload on a given device. Then the device's observed computing capacity, C_{dev} for that problem is given by $C_{dev} = \frac{W}{T_{dev}}$. Identically, if the execution time of that workload on a given heterogeneous set D of devices is T_D, then $C_D = \frac{W}{T_D}$. The heterogeneous speedup, $S_h(D)$, relatively to the execution time on some given single reference device ref (e.g. the slowest) is then given by Equation 3.5:

$$S_h(D) = \frac{T_{ref}}{T_D} = \frac{C_D}{C_{ref}} \tag{3.5}$$

Intuitively, the computing capacity available on the set D of devices is given by the sum of the individual

capacities of all devices in D, i.e., $C_D^* = \sum_{i \in D} C_{dev_i} = W \sum_{i \in D} \frac{1}{T_{dev_i}}$. Heterogeneous efficiency can now be defined as the ratio of used computing capacity over the available capacity:

$$E_h(D) = \frac{C_D}{C_D^*} = \frac{\frac{1}{T_D}}{\sum_{i \in D} \frac{1}{T_{dev_i}}} \tag{3.6}$$

In Section 3.6.2 a strong scalability analysis is performed (constant problem size, i.e., constant W) by using $E_h(D)$ for different heterogeneous sets of devices D. Equation 3.6 shows that if, due to algorithmic and implementation penalties, the used computing capacity, C_D, grows at a lower rate than C_D^* , then $E_h(D)$ will become smaller as the number of devices in D increases.

3.5.3 Computing System

The computing system used to assess the proposed framework is equipped with two Intel Xeon CPU E5649, each running at 2.53GHz with **six cores** and 12GB of memory RAM. The platform is also equipped with **a NVIDIA Fermi GTX 480** with 480 CUDA cores and 1.5 GB of memory, plus **two NVIDIA Tesla C2070**, with 448 CUDA cores with 6GB of memory. The code was compiled with the GNU C compiler 4.6 and NVCC compiler, provided by CUDA toolkit 5.5, in a LINUX operating system.

3.6 Results

This section presents and discusses experimental results with respect to scheduling of irregular workloads, performance scalability and a comparison with a state of the art framework – StarPU.

3.6.1 Scheduling Irregular Workloads

Irregular applications imply unbalanced computational demands across data elements, which, on wide SIMD architectures, would result on severe resource under-utilization. Consumer-producer kernels are thus proposed as the means to avoid this potential performance penalty at the Tier-2 parallelism level. Figure 3.4 presents performance comparisons for the consumer and consumer producer-kernels, labelled as C_kernel and CP_kernel, respectively, for the BH, PT and FL applications with different problem sizes and using a single GPU. Speedup of the consumer-producer kernel over the consumer kernel is also presented in the rightmost axis. Note that PT plot in the middle depicts PT throughput, expressed in MRays/s, instead of execution time. Throughput will be used throughout this work for PT because it provides an abstraction to the light transport model details and algorithms' implementation. A further reason to use throughput is that the performance of PT will be compared to a reference path tracing version using SmallLux (Table 3.3). SmallLux uses a slightly different light transport model that results on tracing different numbers of rays; by reporting rays per second, for the same scene and rendering

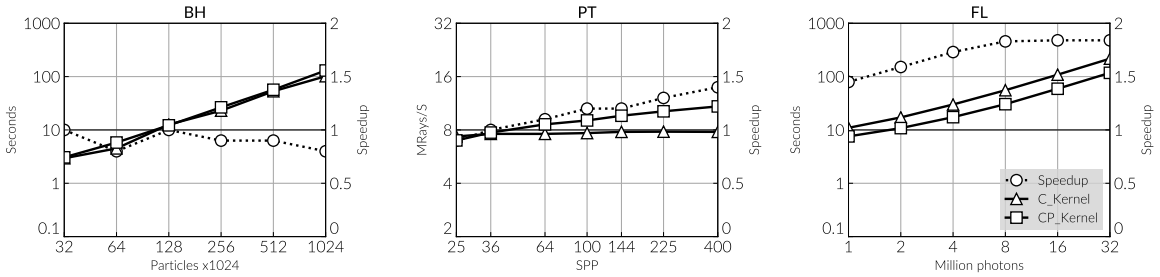


Figure 3.4: Performance comparison between C-Kernel and CP-Kernel on a single GPU. Note the left-handed y-axis and x-axis in log scale and right-handed y-axis in linear scale.

parameters, performance comparisons can be made.

The consumer-producer approach provides a significant speedup for both the pathtracer (40% better) and fluorescence (84% better), while performing about 20% worse in the BH applications. While the basic work unit for the BH consumer-producer kernel consists on a very light task (deciding, for one node of the octree, whether its children have to be visited and computing the resulting force for those that are not), for PT this is a demanding task, requiring tracing a ray and associated shadow rays as well as shading computations. An hypothesis is that the workload associated with each BH basic work unit is not enough to compensate the overheads associated with queuing and scheduling the dynamically generated basic work units. In order to verify this hypothesis a parameterizable synthetic workload is added (SW – computing the Fibonacci sequence up to a given index, whenever an octree node is visited) to the Barnes-Hut consumer and consumer-producer basic work units.

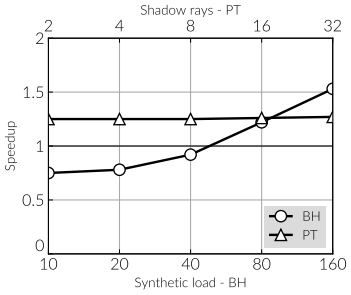


Figure 3.5: Load impact in performance, expressed in terms of speedup of the consumer-producer kernel over the consumer one. Number of shadow rays per shading point in PT (upper horizontal axes) and synthetic load for BH (lower horizontal axes). Note that both horizontal axes are in log scale.

Figure 3.5 depicts the observed speedups for both BH and PT applications – actual values in Table 3.1. Note that in BH, as the SW increases the consumer-producer kernel becomes more effective (maximum of 53% faster) than the consumer kernel, which corroborates the above cited hypothesis. The PT result also corroborates the above conclusions. As the load per basic work unit increases (expressed as the number of shadow rays cast per shading point to assess the visibility of the light sources), speedup increases although at a marginal rate compared to BH (Table 3.1); this is due to the fact that, even with only one shadow ray per point, the load associated with each basic work unit is enough to overcome the

Table 3.1: Speedup of the consumer-producer kernel over the consumer kernel with load impact in performance as workload is increased per BWU in BH and PT.

BH		PT	
synthetic load	speedup	shadow rays	speedup
0	0.75	2	1.25
20	0.78	4	1.25
40	0.92	8	1.25
80	1.22	16	1.26
160	1.53	32	1.27

overheads associated with the queuing system. Given that without synthetic workload the consumer-producer kernel is not effective for BH, results obtained with this application will not be further reported on this Subsection. BH results with the consumer kernel without synthetic workload will be presented in Section 3.6.2 to demonstrate that the proposed framework can still effectively handle this kind of workloads.

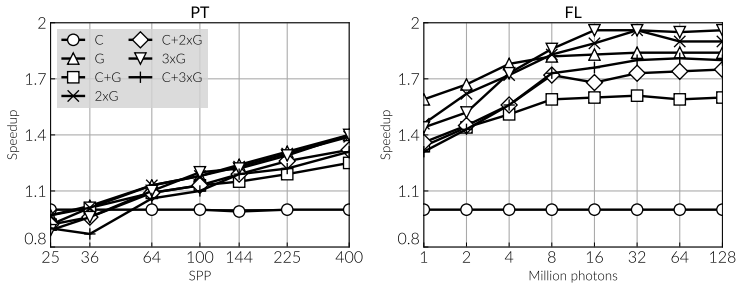


Figure 3.6: Performance comparison between consumer kernel and consumer-producer kernel with multiple-device configurations when scheduling PT and FL irregular workloads. C stands for CPU and G for GPU. Note that horizontal axis is in log scale.

Figure 3.6 shows the speedup obtained with the consumer-producer kernel over the consumer kernel for the PT and FL applications with different configurations of multiple heterogeneous devices and for different problem sizes. For a single GPU the lines are the same as in Figure 3.4. Note that Figure 3.6 tries to illustrate the speedup of using consumer-producer kernel with multi-devices – multi-device (Tier-3) scheduling is assessed in detail in the Section 3.6.2. In PT for multiple-device configurations the achieved speedup increases monotonically with the problem size to a maximum of 1.42x with three GPUs and 400 SPPs. As for the FL case, the speedup increases until a certain workload and then stabilizes with a maximum of 1.96x with three GPUs – see Section 3.6.2 for a discussion on why is the speedup obtained

Table 3.2: Performance values with multi-device configurations. C stands for CPU and G for GPU.

App	Workload	C	G	C+G	2xG	C+2xG	3xG	C+3xG
MM (sec)	7k x 7k DP	12.14	4.16	3.61	2.31	2.16	1.64	1.60
BH (sec)	1024k particles	291.99	101.60	80.26	58.11	55.09	42.08	37.64
PT (MRays/sec)	400 SPP	5.39	10.86	13.43	19.16	23.82	27.03	30.63
FL (sec)	32M Photons	542.32	120.26	100.47	66.66	60.82	46.09	44.23

with FL significantly larger than that of PT. These results clearly show that the consumer producer kernel provides a clear gain over the consumer approach, and that this gain is sustainable in the presence of multiple heterogeneous devices. Also note that using this multiplicity of heterogeneous devices requires no additional programming effort from the application developer, which is this work main goal.

Consumer vs Consumer-Producer

The application programmer is responsible for selecting whether a consumer or a consumer-producer kernel is used to implement a given job. A consumer kernel has the advantages of allowing the utilization of optimized third party libraries and having an associated execution and programming model familiar to most programmers. A consumer-producer kernel explicitly handles load imbalances within a device, but exhibits overheads associated with queue management. The latter should be preferred over the former whenever the application workload is expected to be irregular, in the sense that it exhibits unpredictable workload and memory access patterns, which vary across elements of the data domain, and the workload per basic work unit mitigates the queue management overhead. In situations where irregular applications do not fulfil this last condition, the consumer kernel can be effectively used instead, which will be demonstrated in the next section.

3.6.2 Performance Scalability

The goal of the proposed framework is to allow efficient execution of irregular data parallel applications while maintaining high programming productivity by hiding from the programmer many of the details associated with such systems; this is achieved by complying with the proposed programming and execution model.

Figure 3.7 presents the performance gain for the selected applications executing on increasing numbers of computational devices – actual values shown in Table 3.2. Since it is a regular application, the consumer

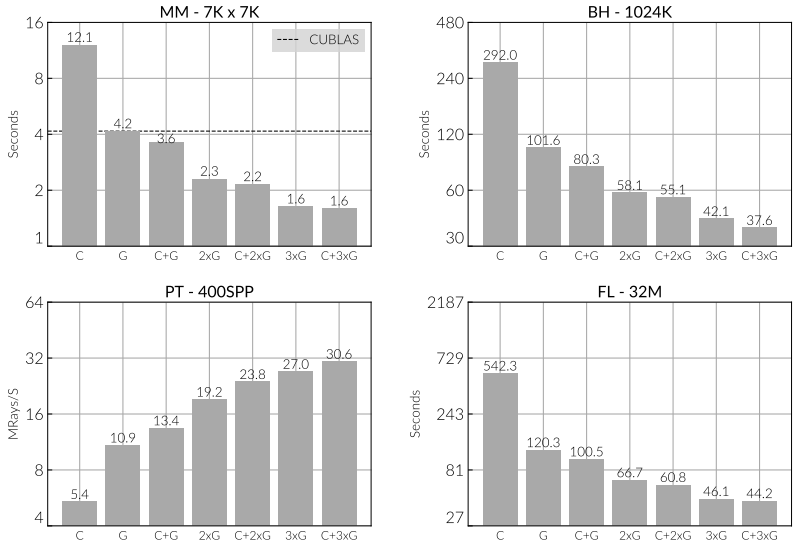


Figure 3.7: Performance with multiple-device configurations. A consumer kernel type is used for the MM and BH applications and a consumer-producer kernel in PT and FL. C stands for CPU and G for GPU. Note the vertical axis in log scale.

Table 3.3: Performance values with multi-device configurations compared to a reference version running on a single GPU. PT values differ from Table 3.2 because a single shadow ray was used per shading point. C stands for CPU and G for GPU.

App	Workload	C	G	Ref (G)	C+G	2xG	C+2xG	3xG	C+3xG
MM (sec)	7k x 7k DP	12.14	**4.16**	**4.16**	3.61	2.31	2.16	1.64	1.60
PT (MRays/sec)	400 SPP	5.03	**12.91**	**12.15**	14.42	22.25	25.96	32.23	34.27

kernel is used for the matrix multiplication. Also, since the consumer-producer kernel is not able to provide performance gains with respect to the consumer kernel for BH, due to the very light workload associated with each BWU, results are reported using the consumer kernel; the goal is to verify whether performance gains are still obtained as the number of heterogeneous devices increases. The consumer-producer kernel is used for the irregular PT and FL applications.

The MM plot clearly shows that the regular matrix multiplication has increased performance as more devices are added. The horizontal dashed line depicts the execution time of the same problem on a single GPU using a reference version developed using CUBLAS (the same library used within the framework provided kernel); there is no performance penalty associated with using this framework for a single GPU and there is a clear gain as more devices are added to the system, since performance scales without any programmer effort (see Tables 3.2 and 3.3). With the four devices working together, the runtime system is able to extract about 8x speedup compared to the single (multicore) CPU configuration.

The BH application plot clearly shows that the execution time decreases as more devices are added,

achieving a maximum 7.6x speedup compared with the CPU configuration (see also Table 3.2). Considering that the consumer kernel is being used for this highly irregular application, this result shows that consumer kernels can still be used effectively to handle irregular workloads. This is particularly useful when an application would exhibit a very light workload per BWU under a consumer-producer model, insufficient to compensate the associated overheads. In such cases, the consumer kernel can still be used and performance will still increase with the number of devices.

The PT plot depicts the PT throughput, expressed in MRays/s, and clearly shows that performance increases significantly as devices are added to the system (the vertical axis is in log scale). Table 3.3 compares the achieved performance with that obtained with a reference single GPU pathtracer based on SmallLux. Note that the values reported for PT are slightly different from those reported on Table 3.2 because now a single shadow ray is being shot per shading point, whereas previously several shadow rays were used. It is clear that the proposed approach suffers no performance penalization compared to the reference SmallLux and that ray throughput increases with the number of devices.

Finally, for the FL application a similar result is achieved, with performance increasing with the number of devices and achieving a remarkable speedup of 12.26x to the single CPU. These larger performance gains obtained with FL when compared to PT result from the minimal memory management overheads associated with the former (as explained in Section 3.5.1) and a large gain when using GPUs compared to the CPU (according to Table 3.2 the GPU is 4.5x faster than the CPU for FL and only 2x faster for PT). A 1.8x speedup can also be observed when adding a Tesla C2070 to a GTX480 (the Tesla has one less SM) and 1.45x speedup when adding another Tesla to the GTX480+Tesla configuration (additional tests were performed that reveled a 1.99x speedup from one Tesla to 2xTesla) – overheads associated with increasing the number of devices are thus minimal for the FL case.

Performance scalability is achieved with minimum programmer effort: adding devices with the same architectures only requires registering them through the HCP, while adding devices with different architectures (supported by the framework through the device API) requires providing the respective kernels. Programming productivity is thus preserved, while enabling efficient execution of regular and irregular applications on heterogeneous systems.

In order to measure how effectively the proposed framework uses the resources available on the parallel heterogeneous system a strong scalability analysis is performed using the heterogeneous efficiency metric introduced in Section 3.5.2. Strong scalability analysis entails studying how the system efficiency varies with the number of devices for a fixed workload (i.e., problem size). Efficiency is expected to decrease with the number of devices, since overheads (such as devices' idleness due to load imbalances, communication and runtime system management costs) increase. However, if efficiency decreases in a very sublinear manner, the system is deemed scalable for fixed problem size. Ideally, the above mentioned overheads would be measured directly; this is however not possible, since multiple management operations occur concurrently and asynchronously. Efficiency analysis provides thus a robust tool to assess the impact of

Table 3.4: Strong scalability: heterogeneous efficiency for the four case studies. 7k x 7k matrix for MM, 1024k particles in BH, 400 SPP for PT and 32M photons in FL. C stands for CPU and G for GPU.

Application	C+G	2xG	C+2xG	3xG	C+3xG
MM	94%	92%	83%	85%	74%
BH	94%	97%	85%	92%	91%
PT	83%	99%	97%	98%	93%
FL	98%	99%	97%	99%	95%

such overheads.

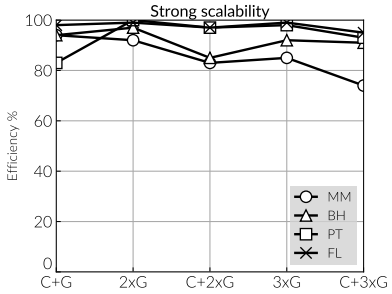


Figure 3.8: Strong scalability: heterogeneous efficiency for the four case studies. 7k x 7k matrix for MM, 1024k particles in BH, 400 SPP for PT and 32M photons in FL.

Figure 3.8 illustrates the variation of heterogeneous efficiency for the four applications with different devices' configurations – Table 3.4 shows the corresponding values. These results show that high efficiency values (above 80%) are maintained for all applications. MM exhibits slightly lower efficiency values than the others because it has a very low computation–communication ratio, i.e., the number of arithmetic operations performed per byte read from memory is very low. There is also a drop in efficiency every time a CPU is added to a multiple GPU configuration. This happens because the CPU exhibits a much lower computing capacity (in the terms defined in Section 3.5.2) than the GPUs for these applications, as can be clearly seen in Figure 3.7 by comparing the C and G bars. It becomes thus extremely difficult for the runtime system to maintain the same efficiency level when a relatively less powerful device is added – remember however that this does not represent a loss in performance for the general case, just a loss in efficiency. Note that the efficiency reported for BH is lower than for PT and FL; however, the consumer kernel is being used for this irregular application. These results confirm that the conclusions drawn above with respect to irregular applications with light workloads per BWU: the consumer kernel can still be used, even though efficiency values will be lower than for more adequate irregular workloads.

Heterogeneous efficiency for all four case studies across different workloads and number of devices is

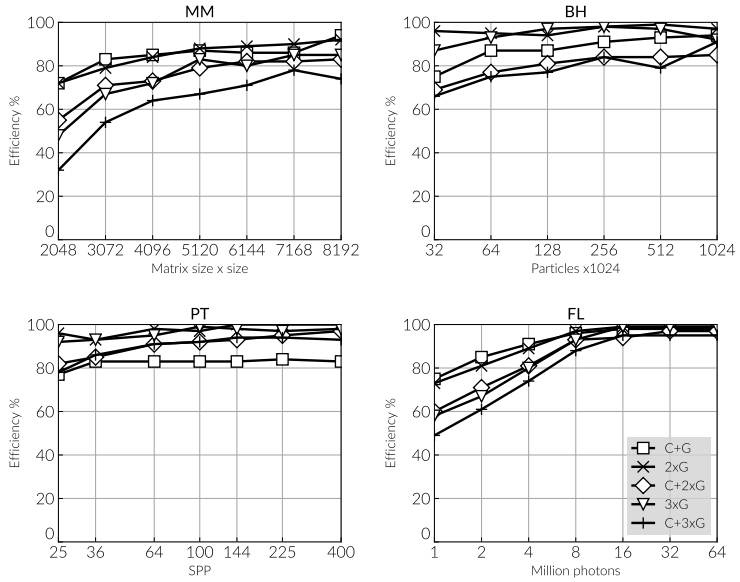


Figure 3.9: Heterogeneous efficiency with multiple workloads and multiple-device configurations. Consumer kernel for MM and BH, consumer-producer kernel for PT and FL. C stands for CPU and G for GPU.

depicted in Figure 3.9. In the general case efficiency increases with the workload and values within the range of 80% to 100% are achieved for the maximum tested workloads. It can thus be concluded that the proposed approach scales well with problem size within the range of devices and workloads evaluated. In the general case efficiency decreases as the number of devices increases, particularly when the CPU is added to a configuration based only on GPUs. This is strong scalability and has been discussed before; overheads are expected to increase with the number of devices and the CPU contributes with a reduced computing capability compared to the GPUs, making it harder to maintain very high efficiency levels. Efficiency, nevertheless, drops sublinearly with the number of devices.

PT and FL achieve higher efficiency than MM and BH across a wide range of problem sizes, with FL still struggling for smaller workloads. MM presents the worst efficiency values and its scalability is the poorest across both dimensions: workload and number of devices. This is due to the low computation-communication ratio. However, it still exhibits an average 80% efficiency and for the highest workload efficiency ranges from 74% to 94%, which are reasonable values considering the memory access overheads. BH's efficiency ranges between 80 and 98% for all problem sizes, except for very small workloads, with maximum values being achieved with two and three GPUs. This is a very positive result since a consumer kernel is being used for an irregular application, given that BH exhibits very low workload per BWU. PT consistently achieves efficiency values above 80% for all workloads and system configurations. Even at low workloads PT performs well given the workload associated with each BWU and in spite of the memory management costs associated with tasks assignment as described in Section 3.5.1. Finally, FL has very low memory management overheads which enables the system to achieve an average of 98% efficiency above 8 million photons. This is very close to the ideal case, demonstrating that with the

proper amount of work to suit available computer power and in the absence of implementation penalties (such as dynamic memory allocation per task), the overhead of the framework is properly compensated by the gains obtained with an effective intra-device scheduling.

3.6.3 Comparison with StarPU

In order to further validate the approach, a comparison of the proposed runtime system with a state of the art heterogeneous system scheduling framework – StarPU [52] – is provided. Both runtime systems have similar data-management mechanisms, but StarPU does not explicitly target irregular workloads, uses a different inter-device scheduling strategy and ignores intra-device scheduling. StarPU scheduling is based on the Heterogeneous Earliest Finish Time (HEFT) algorithm [53] and in a history-based performance modelling. The HEFT has demonstrated to achieve good results with regular workloads on heterogeneous systems, but it does not address irregular workloads. We implemented the PT application in StarPU using the typical algorithm equivalent to C_kernel and compare with the proposed runtime system using the consumer-producer execution model. In StarPU, it is the user's responsibility to specify the task granularity, therefore multiple grain sizes were tested and selected the one achieving the best results (240 tasks for most of the device configurations).

Figure 3.10 illustrates the speedup of the proposed approach over StarPU with multiple device configurations and different workloads. With a single multi-core CPU the framework achieves a fairly constant speedup of 1.30x. The different tasks' sizes in both frameworks results on different behaviours that justify this speedup. The remaining configurations clearly show the benefit of using intra-device scheduling mechanisms. With a single GPU a consistent increase in speedup is observed up to 1.53x. Adding a CPU reduces the speedup because the gain with the CPU is lower and constant, but for the remaining configurations the speedup increases consistently achieving a maximum of about 1.50x with 576 SPP. The persistent kernel approach is able to balance the load within the GPU, which increases resource utilization and also leverages the coherence exhibited by the algorithm. These results clearly show that the proposed approach consistently achieves larger performance than StarPU for irregular workloads and that this performance gain increases with the workload size, thus favoring larger problem sizes. Also, even though speedups are reported only for up to 4 devices (one multi-core CPU and three GPUs), the data suggests, specially for larger workloads, that no inflection point is about to be reached and that additional devices would still exhibit significant speedups over StarPU. This conclusion has to be validated once access to a system endowed with more computing devices is available. Combined with a suitable and unpredictability tailored inter-device scheduling the proposed approach is thus able to deliver more performance and to efficiently exploit the available computing resources when compared with a state of the art system designed for regular workloads such as StarPU.

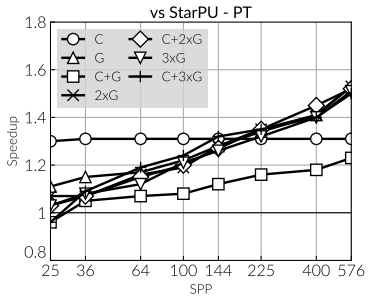


Figure 3.10: Path tracing – Speedup of the proposed approach over StarPU with multiple device configurations when scheduling irregular workloads. C stands for CPU and G for GPU. Note that horizontal axis is in log scale

3.7 Conclusions and Future Work

This contribution presents a framework for efficient execution of data parallel irregular applications on heterogeneous systems while maintaining high programming productivity. The Tier-3, Tier-2 and Tier-1 parallelism levels are addressed. The framework integrates a unified programming and execution model with data-management and scheduling services, that keep the programmers agnostic to HS particularities, allowing them to concentrate on the application functionality.

Part of the results concentrate on the programming model and on Tier-3 scheduling showing that both regular and irregular applications scale well as more devices are added to the computing system. They also show that Tier-2 and Tier-1 scheduling, based on consumer-producer kernels, is able to sustain significant performance gains over consumer kernels for irregular applications, as long as the workload per basic work unit is enough to compensate the overheads associated with queuing and scheduling the large number of dynamically generated tasks. If the application exhibits a very low workload per basic work unit, then consumer kernels can still be used.

The proposed framework has proven to enable efficient exploitation of HS for irregular applications, while requiring minimal programming effort: using additional devices with architectures already exploited by the application only requires registering them through the HCP, while adding devices with different architectures (supported by the framework through the device API) requires providing the respective kernels. Expanding the framework support to new device architectures requires developing API implementations for those architectures, a task to be entailed by the framework developers, not application programmers. The runtime system was further validated and compared with a heterogeneous system (Tier-3) scheduling framework – StarPU. Results reveal that our approach is able to outperform a state of the art runtime system designed for regular workloads. This is, to the best of our knowledge, the first published integrated approach that successfully handles irregular workloads over heterogeneous systems.

The future work in this contribution includes extending the proposed framework to support other archi-

tectures such as DSPs and Intel PHI's, and to further assess the scalability of the proposed mechanisms with systems with a larger number of devices.

Chapter

4 Heterogeneous Distributed Systems

Contents

This chapter describes an approach to tackle the challenges posed by multi-node heterogeneous systems. The approach is essentially based on a dynamic load balancing approach, designed to handle dynamic workloads in systems with performance imbalances across computing nodes. The approach is integrated into a widely used numerical simulation library and evaluated in multiple systems with different imbalance levels.

4.1 Introduction

The contribution discussed in this chapter will address **Tier-4 parallel** computing systems. These systems, typically known as clusters or supercomputers, are composed by multiple nodes connected by a network interface in a distributed memory layout. Clusters are one of the most widely available parallel systems and provide a cost-effective, extensible and powerful computing resource. One of the most important branch of applications executed in these systems are CFD simulations which will be the main target of this contribution.

CFD simulations have become a fundamental engineering tool, witnessing an increasing demand for added accuracy and larger problem sizes, being one of the most compute intensive engineering workloads. The most common approaches to CFD, such as Finite Elements (FEs) and FVs, entail discretizing the problem domain into cells (or elements) and then solving relevant governing equations for the quantities of interest for each cell. Since each cell's state depends on its neighbours, solvers employ some form of nearest neighbour communication among cells and iterate until some convergence criteria are met. Typically, CFD problems are unsteady, requiring an outer loop which progresses through simulation time in discrete steps. Domain decomposition is used to make available a suitable degree of parallelism, i.e., the set of discrete cells is partitioned into subsets which can then be distributed among the computational resources. Such very compute intensive type of workloads are obvious candidates to exploit the inherent parallel computing capabilities of Tier-4 systems.

These systems can be fairly easily extended by adding more nodes with identical architectures, but often from newer generations offering more computing capabilities. This extensibility renders the system **heterogeneous** in the sense that different generations of hardware, with diverse configurations, coexist in the same system. An additional source of heterogeneity is the integration on current supercomputing clusters [23] of devices with alternative architectures, programming and execution models, such as the new highly parallel Intel KNLs and the massively parallel GPUs [64].

However, this heterogeneity results in **different performances** across nodes, potentially leading to severe load imbalances. Static and uniform workload distribution strategies, as typically used by CFD software, will result on the computational units waiting on each other and resources underutilization. Properly distributing the workload and leveraging all the available computing power is thus a crucial feature, which

has been revisited in the latest years due to increasing systems' heterogeneity [65].

The load distribution problem is further aggravated in the presence of dynamic workloads. CFD solvers often refine the problem domain discretisation as the simulation progresses through time, allowing for higher accuracy in regions where the quantities of interest exhibit higher gradients. In the scope of this work, these applications will be referred to as *adaptive applications*. This refinement entails splitting and merging cells, resulting on a new domain discretisation. Given that the computational effort is in general proportional to the number of cells, its distribution across the problem domain also changes. Not accounting for this refinement and maintaining the initial mapping throughout the whole simulation would lead to load imbalances and huge performance losses.

The combination of the differences in computing power provided by the heterogeneous CUs with the differences in computing requirements from dynamic workloads, defines one of the main challenges identified in this thesis — the *two-fold challenge* (Section 1.1.2). The adoption of DLB is proposed as a means to address this computing imbalance as a whole and allows for fully leveraging all the available computing power and improve execution time. The proposed mechanisms in this chapter will address the particularities of Tier-4 parallelism and target the most impacting challenge on these systems — the load and performance imbalances.

This contribution will thus focus in **combining DLB** with HS in the context of CFD simulations by integrating DLB mechanisms in a widely used application: **OpenFOAM**. OpenFOAM is a free and publicly available open-source software package, specifically targeting CFD applications [66]. It is an highly extensible package, allowing applied science experts to develop scientific and engineering numerical simulations in an expedite manner. OpenFOAM includes a wide range of functionalities such as simulation refinement, dynamic meshes, particle simulations, among others. OpenFOAM large set of features and extensibility has made it one of the most used and leading open-source software packages across the CFD community. It has also been made available in multiple supercomputers and computing centres, along with technical support. OpenFOAM parallel distributed memory model is based on a domain decomposition approach, however, there is little to no support for either HS or DLB, which is addressed by this work by integrating and evaluating all proposed mechanisms into this package. More details on OpenFOAM are presented below.

Providing such support is of crucial importance, however, this task is too complex to be handled by the CFD application developer. This complexity has two different causes: i) efficient mapping of the dynamic workload onto a vast set of heterogeneous resources is a research level issue, far from the typical concerns of a CFD expert, and ii) execution time migration of cells (particularly dynamically refined meshes of cells) across memory spaces requires a deep understanding of OpenFOAM's internal data structures and control flow among lower level code functions and methods. Integration of these facilities with OpenFOAM by computer science experts is proposed as the best solution to provide efficiency and robustness, while simultaneously **promoting reuse by the CFD community**.

The OpenFOAM Challenge

Open Source Field Operation and Manipulation (OpenFOAM) is a powerful C++ software package developed for CFD and other multi-physics engineering problems. The library addresses the three main stages of a numerical simulation (pre-processing, solving and post-processing) and it is centred on the concept of applications that are subdivided in solvers and utilities. Specific solvers are developed to solve a particular continuum mechanics problem, while utilities are mainly related to data manipulation and analysis.

OpenFOAM, originally known as FOAM, was created in the Imperial College London, however, its development path suffered from **severe fragmentation resulting in multiple development parties and forks**. Its development started in the later 1980s and in 2000, Henry Weller together Hrvoje Jasak, founded Nabla Ltd as the main development party. In 2004, the team diverged, and Weller founded OpenCFD Ltd. Simultaneously, Jasak founded Wikki Ltd and developed a fork — foam-extend. In 2011, OpenCFD was acquired by SGI and then by the ESI Group in 2012. Two years later, Weller left the ESI Group and continued the development at CFD Direct Ltd on behalf of the OpenFOAM Foundation Ltd, to which the copyright of OpenFOAM was transferred at some point. The maintained and distributed forks are thus the **CFD Direct**, **ESI Group** and the **foam-extend** forks.

OpenFOAM is considered the most used free CFD library and it is in the top 5 of the most used CFD libraries. It has an active development and support with several reported issues being submitted and resolved per week. OpenFOAM is a complex software package with over **1.5 million lines of code** scattered over about **half a million files**. It makes full use of C++ object inheritance and polymorphism features, together with C++ templates. The parallel execution approach is based on the distributed memory model using MPI and domain decomposition. Two levels of the development can be identified: (i) Solver development, where new solvers are developed or adapted and (ii) core development, related to the development of the OpenFOAM core functionality. Given the complexity of the package, the latter is far more challenging, requiring a deeper understanding of the whole architecture of the library. Its open-source development approach also contributes to its complexity as a large percentage of the components were developed by a variety of programmers and applied experts.

Another challenging aspect of the development, not only of OpenFOAM but any CFD software package, is the **inherent behaviour of fluid mechanics and their simulation**. For instance, convergence is a major issue in CFD simulations that requires knowledge and insight on the specific physical phenomena being simulated. Any change applied to a simulation code, particularly parallel code, may promptly result in a non-convergent simulation. A non-convergent simulation will not only provide incorrect physical results, but it may also result in residual overflow and/or unstable code execution.

To approach the above hypothesis, this work proposes **nSharma — Numerical Simulation Heterogeneity Aware Runtime Manager** — a runtime manager that provides OpenFOAM with heterogeneity aware DLB features. nSharma monitors the heterogeneous resources performance under the current load, combines this data and past history using a performance model to predict the resources behaviour under new workload resulting from the refinement process and makes informed decisions on how to re-distribute the workload. The aim is to minimize performance losses due to workload imbalances over HS, therefore contributing to minimize the simulation's execution time. DLB minimizes idle times across nodes by progressively and in an educated way assigning workload, which can be itself dynamic, to the available resources. nSharma package integrates in a straightforward manner with current OpenFOAM distributions, enabling the adoption of heterogeneity aware DLB. To best of author's knowledge, this is the first implementation and integration of heterogeneous-aware DLB mechanism in OpenFOAM.

4.2 Related Work

Libraries supporting the development of CFD simulations, include **OpenFOAM**[66], **ANSYS Fluent**[67], **ANSYS CFX**[68], **STAR-CCM+**[69], among others. OpenFOAM is distributed under the General Public Licence (GPL), allowing modification and redistribution while guaranteeing continued free use. This motivated the selection of OpenFOAM for the developments envisaged in this work. The authors see no reason why this document's higher level assessments and results can not be applied to other similar CFD libraries. This generalization should, however, be empirically verified on a per case basis.

Domain decomposition requires that the mesh discretization is partitioned into sub-domains. This is a challenging task impacting directly on the workload associated with each sub-domain and on the volume of data that has to be exchanged among sub-domains in order to achieve global convergence. Frameworks that support mesh-based simulations most often delegate mesh partitioning to a third-party software. **ParMETIS** [70] and **PTSCOTCH** [71] are two widely used mesh partitioners, which interoperate with OpenFOAM. ParMETIS has been used within this work's context because it provides a more straightforward support for Adaptive Mesh Refinement (AMR).

ParMETIS includes methods to both partition an initial mesh and re-partition a mesh that is scattered across CUs disjoint memory address spaces, avoiding a potential full re-location of the mesh in runtime. The (re)partitioning algorithms optimize for two criteria: minimizing edge-cut and element migration. These criteria are merged into a single user-supplied parameter (ITR), describing the intended ratio of inter-process communication cost over the data-redistribution cost. ParMETIS also provides an interface to describe the relative processing capabilities of the CUs, allowing more work units to be assigned to faster processors .nSharma calculates these parameters in order to control ParMETIS' repartitioning and thus achieve efficient DLB.

Some frameworks providing DLB to iterative applications have been proposed. **DRAMA** [72] provides a collection of balancing algorithms that are guided by a cost model which aims to reduce the imbalance costs. It is strictly targeted for finite element applications. **PREMA** [73] is designed to explore an over-decomposition approach to minimize the overhead of stop-and-repartition approaches. This approach is not feasible in some mesh-based numerical simulations (due to, for instance, data dependencies) and no mention to HS support could be found. **Zoltan** [74] uses callbacks to interface with the application and integrates with **DRUM** [75], a resource monitoring system based on static benchmark measured in MFLOPS and averaged per node. The resource monitoring capabilities of nSharma are much more suitable to account for heterogeneous computing devices – see next section. Zoltan is not tied to any particular CFD framework. It does not enforce any particular cost functions and uses abstractions to maintain data structure neutrality. This however comes at the cost of requiring the CFD application developer to provide all data definitions and pack/unpack routines, which in a complex application like

OpenFOAM is an programming intensive and error prone task.

nSharma integrates with OpenFOAM, accessing its data structures and migration routines. Although this option implies some code portability loss (across alternative libraries), it avoids the multiple costs of data (and even conceptual) transformations together with overheads of code binding between different software packages. This allows direct exploitation, assessment and validation of DLB techniques for OpenFOAM applications on HS. The results on conceptually more abstract design options, such as the performance model and the decision making mechanism, should still generalise to alternative software implementations, although empirical verification is required.

Some of the above cited works can handle HS. They do so by using high-level generic metrics, such as vendor announced theoretical peak performances or raw counters associated to generic events such as CPU and memory usage [75], [76]. The associated performance models are however generic, ignoring both the characteristics of CFD workloads and emerging devices particular execution models and computing paradigms, and thus tend to be inaccurate [77]. This work proposes a performance model which explicitly combines the workload particularities with the heterogeneous devices capabilities. The design of this performance model is strictly coupled with the requirements of the proposed DLB mechanisms.

FuPerMod [78] explores Functional Performance Models, extending traditional performance models to consider performance differences between devices and between problem sizes. It is based on speed functions built based on observed performances with multiple sizes, allowing the evaluation of a workload distribution [77]. Zhong applied these concepts to OpenFOAM [79] and validated it in multi-core and multi-GPU systems. This contribution introduces a similar performance model tightly integrated with the remaining DLB mechanisms.

Mooney et al. [80] addressed AMR in OpenFOAM and proposed a simple approach to perform automatic load balancing on homogeneous systems and directly integrated in OpenFOAM. The work focused on moving boundaries and re-meshing and presented some initial results. Because OpenFOAM does not support migration of refined meshes, Mooney et al. also proposed and implemented a mechanism to enable such migration. This mechanism is used in this contribution as discussed in the following sections.

4.3 nSharma's Architecture

OpenFOAM simulations are organized as *solvers*, which are iterative processes evaluating, at each iteration, the quantities of interest across the problem domain. Each iteration includes multiple inner loops, solving a number of systems of equations by using iterative linear solvers. Within this work, *solver* refers to OpenFOAM general solvers, rather than the linear solvers. Since OpenFOAM parallel implementation is based on a zero layer domain decomposition over a distributed memory model, the solver's multiple processes synchronize often during each iteration, using both nearest neighbour and global communica-

tions.

nSharma is fully integrated into OpenFOAM and organized as a set of components, referred to as modules or models. The **Online Profiling Module (OPM)** acquires information w.r.t. raw system behaviour. The **Performance Model (PM)** uses this data to build an approximation of each CU performance and to generate estimates of near future behaviour, in particular for different workload distributions. The **Decision Model (DM)** decides whether workload redistribution shall happen, based on this higher level information and estimates. The **Repartitioning Module (RM)** handles the details of (re)partitioning subdomains for (re)distribution across multiple processors, while finally load redistribution mechanisms carry on the cells migration among computing resources, therefore enforcing the decisions made by nSharma.

The whole DLB mechanism is tightly coupled with OpenFOAM iterative execution model. This allows nSharma to learn about system behaviour and also allows for progressive convergence towards a globally balanced state - rather than trying to jump to such a state at each balancing episode. Dynamic workloads are also handled by OpenFOAM and nSharma iterative model, with impact on the whole system balanced state and simulation execution time being handled progressively.

Note that the runtime is fully integrated in the OpenFOAM software package and distributed as a plug-in. The mechanisms introduced operate transparently, meaning that no action is required to the OpenFOAM end-user apart from some parametrization. This way, nSharma enables the use of DLB in HS with no effort, substantially increasing productivity which is one of the main challenges identified in this thesis.

4.3.1 Online Profiling Module

The *OPM* instruments OpenFOAM routines to measure execution times, crucial to estimate the CUs relative performance differences. This has been achieved by thoroughly analysing OpenFOAM workflow and operations, and identifying a set of low-level routines that fundamentally contribute to the application execution time. It has been empirically verified that these times correlate well with the computational effort, enabling nSharma to monitor only the parts of the simulation that are relevant to the associated performance modelling. The selective profiling nature also allows for a low instrumentation overhead without any additional analytical models or benchmarking.

The procedures are registered and measured using a simple API that defines two types of procedures: a **Section** and an **Operation**. Sections represent a block procedure, e.g. solve pressure equation, and they may contain multiple other Sections, Operations, synchronizations, etc. Operations exist within Sections and represent the lowest level procedure. There are two types of Operations: an **IDLE** type Operation which is a synchronization or a memory transfer, and a **BUSY** type Operation that represents a computational task without any synchronizations or network communications. Each CU will measure its own routines and upon central request, will only send pertinent information to a master entity. Each CU will also compute the accumulated time for each BUSY operation, required for the model. This

categorization of execution time allows to measure performance individually, otherwise execution time would be cluttered by dependencies and communications.

4.3.2 Performance Model

The *PM* characterizes the system's – and its individual components, such as each CU – performance and provides estimates of future performances under different workload distributions. Workload and performance characterization requires the definition of a work unit, upon which problem size can be quantified. OpenFOAM uses Finite Volumes, with the problem domain discretisation being based on cells that are combined to define the computational domain. With this approach problem size is often characterized by the number of cells, which is, therefore, the work unit used by nSharma.

Each CU performance is characterized by the average time required to process one work unit, denoted by r_p (where p indexes the CUs). For each iteration i and CU p, the respective performance index (r_p^i) is given by the ratio of the iteration's total busy time over the number of cells assigned to p, N_p^i:

$$r_p^i = \frac{\sum_j^B \beta_p^{j,i}}{N_p^i} \tag{4.1}$$

where N_p^i is the number of cells assigned to CU p and $\beta_p^{j,i}$ is the busy time for each operation j from the set of operations B captured by the OPM. The actual metric used for balancing decisions, $\tilde{r}_p^i$, is a weighted average over a window of previous iterations, which smooths out outliers and, for dynamic workloads, takes into account different problem sizes (different numbers of cells assigned to each CU at each iteration).

To estimate the execution time of the next iteration, T_p^{i+1}, with an arbitrary number of cells, N_p^{i+1}, the PM uses the the above described metric multiplied by N_p^{i+1}:

$$T_p^{i+1} = \tilde{r}_p^i \times N_p^{i+1} \tag{4.2}$$

4.3.3 Decision Module

It is the *DM* role to assess the system balancing state and decide whether a load redistribution step should take place. It is also the *DM* who decides what load to redistribute. Assessing and making such decision is referred to as a *balancing episode*. Since these episodes represent an overhead, it is crucial to decide when should they occur. nSharma allows them only at the beginning of a solver iteration, and defines a period, expressed in number of iterations, for their frequency. The unpredictability of dynamic workloads makes it unpractical to define an optimal balancing period, therefore it is auto-tuned in execution time, as described below.

At the beginning of a new solver's iteration i, the **Relative Standard Deviation (RSD)**, among the CUs busy times for the previous iteration $i-1$ is calculated:

$$\text{RSD}^{i-1} = \frac{\sigma^{i-1}}{\sum_j^B \beta_p^{j,i-1}} * 100 \tag{4.3}$$

standard deviation, σ, is well known as a good, light-weight, indicator of a system's balancing state. A linear regression is then computed over the last few iterations RSD in order to estimate its rate of change, which is used to update the period. Also, a normalization of the magnitude of the RSD is added to the contribution to update the period. Therefore, the load balancing period is adjusted based on how fast the system's balancing state changes and how much it changes.

When a load balancing episode is triggered the *DM* will compute, for each CU p, how many cells, N_p^{i+1}, to assign to it in the next iteration. It will devise a new load distribution, where all CUs will take, the same amount of time to process the assigned work units, according to each CU execution rate, $\tilde{r}_p^i$. Since the total number of cells N is known, **a well-determined system of P linear equations can be formulated** (see Equation 4.4) and solved to find $N_0^{i+1}, \ldots, N_{P-1}^{i+1}$ – the number of cells to assign to each CU.

$$\begin{cases} \tilde{r}_0^i \times N_0^{i+1} = \tilde{r}_1^i \times N_1^{i+1} \\ \tilde{r}_1^i \times N_1^{i+1} = \tilde{r}_2^i \times N_2^{i+1} \\ \quad \ldots \\ \tilde{r}_{P-2}^i \times N_{P-2}^{i+1} = \tilde{r}_{P-1}^i \times N_{P-1}^{i+1} \\ N_0^{i+1} + N_1^{i+1} + \ldots + N_{P-1}^{i+1} = N \end{cases} \tag{4.4}$$

After computing this new distribution, **a decision** has to be made as to whether it will be applied or not, by taking into account the cells migration cost, m. The goal is that the remaining simulation execution time after the load redistribution must be smaller than not migrating. The next iteration $i+1$ expected execution time **without load redistribution** is given by (note that N_p^i and N_p^{i+1} are the same):

$$T^{i+1} = \max_{p \in \{0,\ldots,P-1\}} (\tilde{r}_p^i \times N_p^i) \tag{4.5}$$

whereas with **the new load distribution** it is:

$$T_{dist}{}^{i+1} = \tilde{r}_p^i \times N_p^{i+1} \tag{4.6}$$

(no need for max because $T_{dist}{}^{i+1}$ is approximately the same for all p, according to Equation 4.4). Let n be the number of remaining iterations and δ represent some additional execution overheads independent on workload redistribution. Then the condition:

$$\left(n \times T^{i+1} + \delta\right) > \left(m + n \times T_{dist}{}^{i+1} + \delta\right) \tag{4.7}$$

expresses that migration will only take place if it is expected to reduce the total remaining execution time, while taking into account the cost of actually enforcing the migration m. This cost is estimated by keeping track of the costs of previous migrations and using a linear regression to estimate the cost of any arbitrary decomposition.

$$T^{i+1} > \frac{m}{n} + {T_{dist}}^{i+1} \tag{4.8}$$

Equation 4.8 (a simplification of the condition equation above) makes it clear that a load redistribution should only be enforced if the cost of migrating cells can be properly amortized across the remaining n iterations. Consequently, towards the end of the simulation, as n gets smaller, the cells migration impact on execution times is progressively higher and load redistribution will become proportionally less likely.

4.3.4 Repartitioning Module

nSharma repartitioning module **interfaces with ParMETIS** (see Section 4.2), by carefully parametrising the relevant methods and by extending some functionality. ParMETIS' repartitioning method is used, which takes into account the current mesh distribution among CUs and balances cells' redistribution cost with the new cells' partition communication costs during the parallel execution of the next iterations. The relationship between these two costs is captured by the ITR parameter. nSharma learns this parameter by requesting multiple decompositions with different ITR values in initial iterations, assessing the most effective ones and converging to a single one. Besides ITR, this method also receives a list of each CU relative computing power, given by $\omega_p = N_p^{i-1}/N$, as evaluated by the Decision Module (Section 4.3.3).

OpenFOAM does not natively **support migration of refined meshes**, which required integrating such support (based on Kyle Mooney's approach [80]). Since each refined cell is always a child of a single original (non-refined) cell and since the refined hierarchy is explicitly maintained, partitioning is applied to the original (non-refined) coarse mesh; after partitioning, the refined mesh is considered to perform migration. To ensure that the original non-refined coarse mesh reflects the correct workload, weights for each coarse cell are provided to ParMETIS based on the number of child cells, which will be used by ParMETIS in devising new partitions. The RM also performs **communication topology-aware repartitioning** in order to tackle heterogeneous communications. nSharma maintains a **Communication Graph (CommGraph)**, with nodes representing sets of CUs that share with each others the same communication medium. The RM requests to ParMETIS a higher level partitioning based on the CommGraph nodes, and then further requests a new partitioning for each such node whenever it includes more than one CU. This hierarchical repartitioning leverages ParMETIS boundary minimization mechanisms, potentially reducing slower links communications.

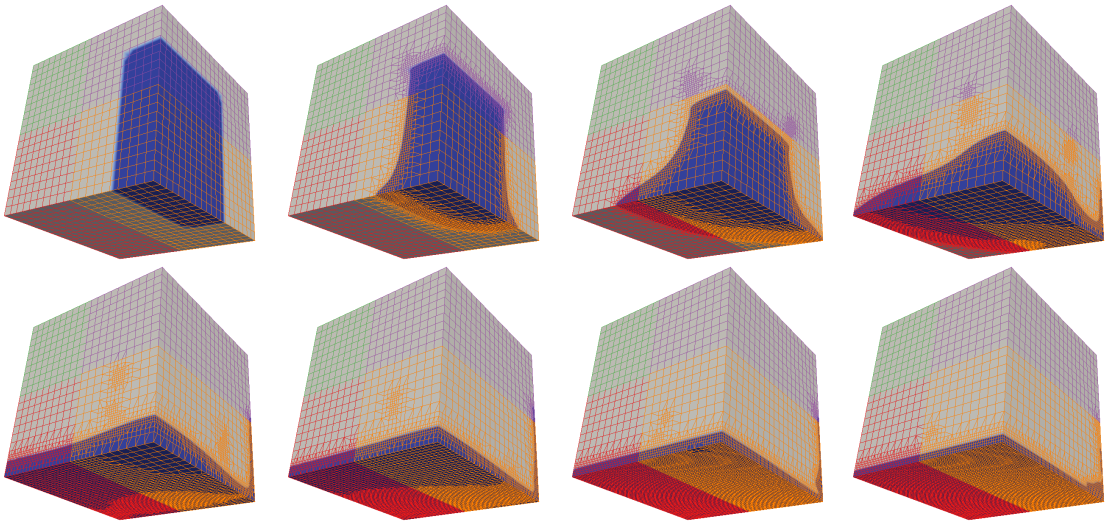


Figure 4.1: *damBreak* geometry and a subset of the simulation result with 4 ranks (each color represents the cells assigned to a different rank) and AMR. Cell distribution devised using ParMETIS and default parametrisation.

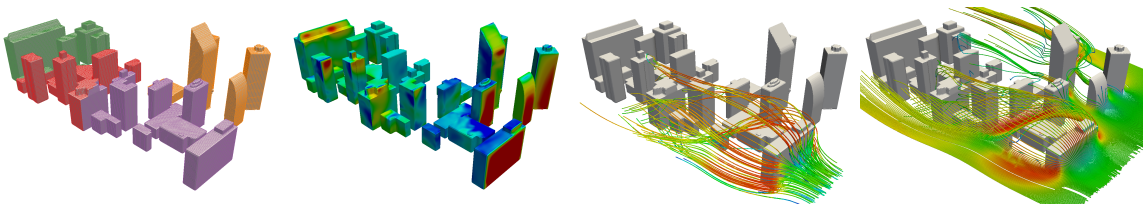


Figure 4.2: *windAroundBuildings* simulation illustration. First plot shows cells distribution over 4 ranks (each color represents the cells assigned to a different rank), second plot illustrates the pressure at time-step 200 and the two last plots show examples of velocity stream lines. Cell distribution devised using ParMETIS and default parametrisation.

4.4 Results

For experimental validation, the ***damBreak*** simulation was selected as the base case study among those distributed with OpenFOAM tutorials. It uses the ***interDyMFoam*** solver to simulate the multiphase flow of two incompressible fluids – air and water – after the break of a column of water driven by gravity. Adjustable time-step was disabled and all other parameters are the same as distributed in the package. For dynamic workloads, AMR subdivides a cell into 8 new cells according to the interface between the water and air; cells will thus be refined (and unrefined) following the evolution of the two phases' interface. Figure 4.1 shows the geometry and a subset of the simulation result with 4 MPI processes (ranks) and AMR. Each colour represents the cells assigned to a different rank and the illustrated cell distribution was devised using ParMETIS and default parametrisation.

Additionally, a fairly different case study was used in order to further validate nSharma capabilities. The ***windAroundBuildings*** simulation, illustrated in Figure 4.2, uses the ***simpleFoam*** solver to simulate the wind

Table 4.1: Computing systems and system configurations used in evaluation

System	SeARCH			Stampede2
Nodes	Tag 641 - Ivy Bridge E5-2650v2 @ 2.60GHz, 16 cores p/node			Tag KNL7250 - Intel Xeon Phi 7250 @ 1.4GHz ("Knights Landing"), 68 cores p/ node
	Tag 662 - Ivy Bridge E5-2695v2@ 2.40GHz, 24 cores p/node			
	Tag 421 - Nehalem E5520 @ 2.27GHz, 8 cores p/node			
	Tag KNL7210 - Intel Xeon Phi 7210 @ 1.3GHz, 64 cores p/ node			
Multi-node configurations	Homogeneous I	Heterogeneous I	Heterogeneous II	Homogeneous II
	Multiple 641's	Pair(s) of 641+421	Pair 662+KNL7210	Multiple KNL7250's
Network	Myrinet (myri)	Myrinet (myri)	Ethernet (eth)	Intel Omni-Path (OPA)

behaviour across a small city composed by multiple different buildings. Pressure and velocity and the main properties assessed by this simulation. Figure 4.2 first plot shows cells distribution over 4 ranks (each color represents the cells assigned to a different rank) — the number of cells is static throughout the simulation, no AMR was applied. The second plot illustrates the pressure at time-step 200 and the two last plots show examples of velocity stream lines. Cell distribution devised using ParMETIS and default parametrisation along with the required changes to compute in parallel.

Note that these solvers require frequent local and global communications. As the degree of parallelism is increased, more sub-domains are created, increasing the number of cells in sub-domains boundaries and, consequently, increasing communications among sub-domains, with network bandwidth and latency impacting significantly in the simulation's performance.

Four hardware configurations were used from two different clusters – **SeARCH cluster** (Universidade do Minho, Portugal) and **Stampede2** (Texas Advanced Computing Center, USA). Configurations are described in Table 4.1. OpenFOAM 2.4.0 was used, compiled with GNU C Compiler in SeARCH and with Intel C Compiler in Stampede2. Each MPI process is associated to one CU, which in this chapter is defined as a processing core: the number of used cores is equivalent to the number of processes. MPI terminology refers to processes as ranks, and this terminology is maintained throughout this section. For the Homogeneous I and Heterogeneous I, the Myrinet network interface is used, however, the Myrinet network cards installed in SeARCH only support up to 8 ports which means that each 641 node is limited to 8 ranks (8 cores).

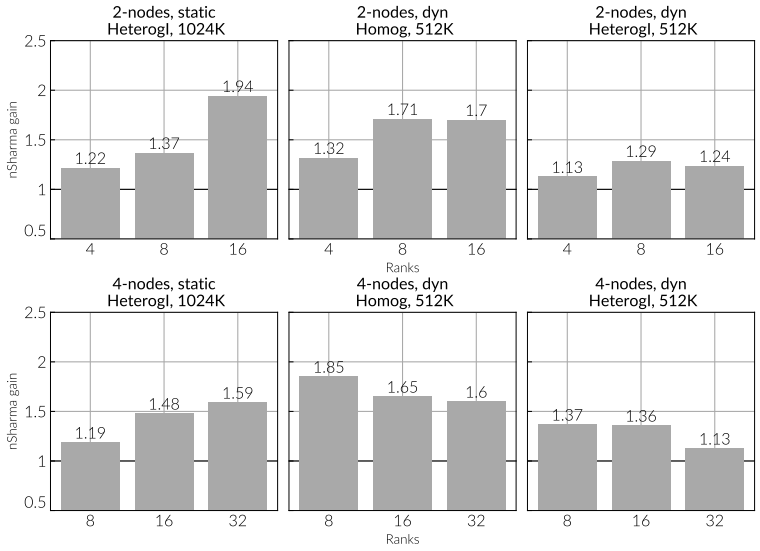


Figure 4.3: nSharma gain with SeARCH Homogeneous and Heterogeneous I

4.4.1 Performance Gain

Performance gain is hereby defined as the reduction in execution time achieved by using nSharma and quantified as the ratio between the execution times without and with nSharma, respectively. Figure 4.3 illustrates such gain for 200 iterations of the damBreak simulation in SeARCH. The first row depicts results obtained with 2 nodes, the second row results obtained with 4 nodes. Results in the first column were obtained with a static workload (no AMR) and problem size of 1024K cells (Heterogeneous I configuration), whereas in the second and third columns dynamic workloads (AMR) were used with 512K cells (Homogeneous and Heterogeneous I configurations, respectively).

nSharma achieves a significant performance gain for all experimental conditions. For static workloads, the gain increases with the number of ranks, with a maximum gain of 1.94 gain with 2 nodes and 16 ranks and 1.59 with 4 nodes and 32 ranks. This gain is basically a consequence of nSharma's heterogeneous awareness, which allows remapping more cells to the 641 more powerful cores, which would otherwise be waiting for the 421 processing cores to finish execution.

For homogeneous hardware and dynamic workloads (second column), performance gain is due to moving cells from overloaded cores to underloaded ones, with such fluctuations due to AMR. Significant gains are still observed for all experimental conditions, but this gain suffers a slight decreases as the number of ranks increases for 4 nodes. This is due to an increase in migration and repartitioning costs (see Figure 4.5), proportional to the increased number of balance episodes required in a dynamic workload scenario (see Figure 4.4). The communication overheads also increase from 2 to 4 nodes sustaining more subdomains and more communications over a limited bandwidth network. In Figure 4.6, an significantly higher number of cells is used, mitigating these overheads and resulting in higher speedup.

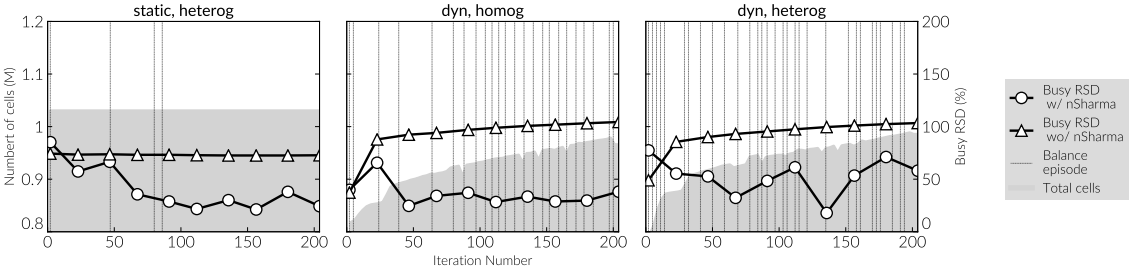


Figure 4.4: Busy RSD with and without nSharma for 4 nodes and 32 ranks.

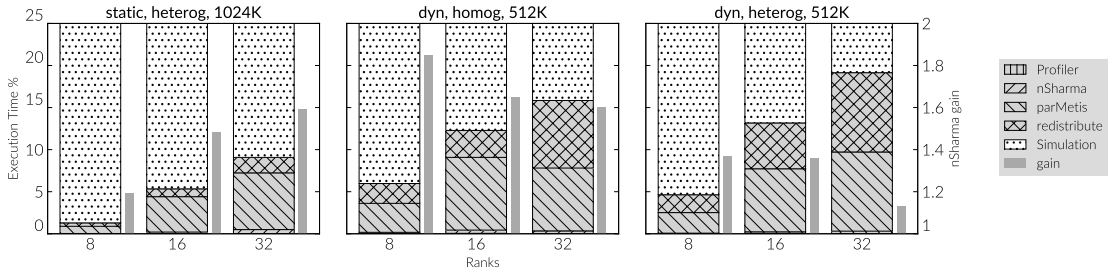


Figure 4.5: Execution time percentage breakdown for 4 nodes

The last column illustrates the combination of dynamic workload with HS. The gain is mostly constant with the number of ranks. It is lower than with static workloads or homogeneous hardware, because the decision making process is much more complex requiring a much higher level of adaptability, i.e more frequent balancing episodes and larger volumes of data migration (see Figures 4.5 and 4.4).

Figure 4.4 illustrates the accumulated busy RSD (as described in Section 4.3.3) with and without nSharma for the same experimental conditions, 4 nodes and 32 ranks. The grey area represents the total number of cells and the vertical lines are balance episodes. Clearly nSharma results in a large RSD reduction, i.e. reduced busy times variation across ranks, thus enabling significant performance gains. This can be clearly seen around iteration number 50 for the static case, where a large RSD reduction occurs.

Figure 4.5 illustrates, for the 4 nodes cases of Figure 4.3, the percentage of execution time spent in different algorithmic segments: *Profiler* represents time used by the OPM, *nSharma* time for decision making, *parMetis* represents repartitioning, *redistribute* is cells migration cost and *simulation* represents the time dedicated to the actual simulation. The side slim bars represent the performance gain, which is the same as in Figure 4.3. The vertical axis goes up to only 25%, the remaining 75% are simulation time and add-up to the illustrated.

The overheads associated with profiling and decision making are negligible in all experimental conditions. Repartitioning (ParMETIS) and redistribution costs increase with the number of ranks. Both exhibit an increasing overhead in all cases which is tightly related to the fact that the numbers of migrated cells and balancing episodes (see Figure 4.4) increase with the hardware configuration and the workload complexities (homogeneous versus heterogeneous and static versus dynamic, respectively). Nevertheless the

overheads associated with DLB are below 17%, allowing for very significant performance gains.

The first three plots of Figure 4.6 presents nSharma performance gain for dynamic workload, 4 nodes, fixed number of ranks and increasing problem size for 3 alternative hardware configurations: SeARCH homogeneous, SeARCH Heterogeneous II and Stampede2 homogeneous (see Table 4.1). Particularly, for Heterogeneous II (662+KNL) configuration, 64 plus 24 ranks are used from KNL and 662 respectively), which corresponds to the use of all available CUs. The performance gain associated with the introduction of DLB increases consistently with the problem size. Larger problems have the potential to exhibit more significant imbalance penalties with dynamic workloads, due to larger local fluctuations in the number of cells. nSharma is capable to effectively handle this increased penalty, becoming more efficient as the problem size increases. Based on the observed data, this trend is expected to continue. No inflection point should be reached and nSharma performance gain will keep increasing with the workload, i.e. exactly when the potential for load imbalances becomes higher.

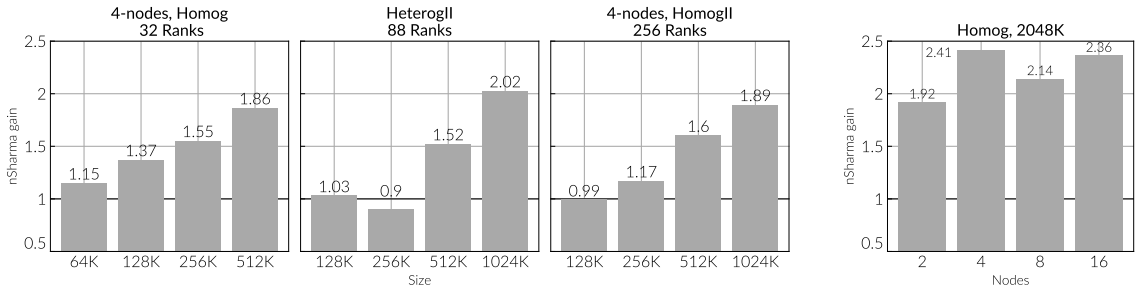


Figure 4.6: First three plots show an increasing problem size for four 641 SeARCH nodes, 662+KNL and four Stampede2 nodes and dynamic workload. Last plot shows an increasing number of 641 nodes using the maximum number of ranks, dynamic workload and about 2 million cells

The last plot of Figure 4.6 shows the performance gain for an increasing number of homogeneous nodes (from 2 to 16 nodes and using the maximum number of ranks) with dynamic workload and about 2 million cells. The gain is substantial – ranging between 1.9 and 2.5x – as nodes increase which provides some insight on the behaviour the nSharma when scaling computational resources as long as the workload is enough to compensate the communication and migration overheads mentioned above. This is an important results since this type of simulations tend to be performed in large scale computing systems.

Figure 4.7 illustrates the results with the *windAroundBuildings* simulation with 4 Heterogeneous I configuration nodes and static workload. This shorter test tries to validate the performance gain of nSharma with a significantly different geometry and workflow, revealing a consistent gain as ranks are increased (between 1.51x to 1.79x), corroborating with the results from the *damBreak* discussed above.

4.4.2 Efficiency Gain

Strong and weak scalability based on parallel efficiency are evaluated in this section. Parallel efficiency is evaluated with respect to the timing results achieved with only 1 rank and without nSharma (DLB is

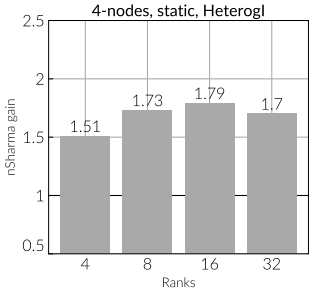


Figure 4.7: *windAroundBuildings* simulation with 4 Heterogeneous I configuration nodes and static workload.

senseless for a single rank).

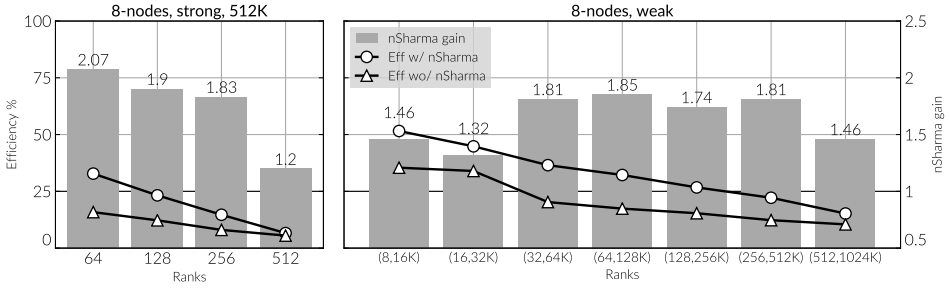


Figure 4.8: Efficiency (w/ and wo/ nSharma) with dynamic loads for Stampede2 nodes

Figure 4.8 presents performance gain with nSharma (bars) and parallel efficiency with and without nSharma (lines), using 8 KNL nodes of Stampede2 (up to 512 ranks). For the strong scaling case – left plot – nSharma performance gain is around 2, except for 512 ranks. In this latter case, the workload per rank is so low (the number of cells ranges from 1000 to 2000 per rank) that incurred overheads (partitioning and cells migration) significantly impact on the load redistribution benefits. For the weak scaling case – right plot – problem size increases at the same rate as number of ranks, thus the workload per rank is kept constant; performance gain is quite consistent, since increasing DLB costs are compensated by the added workload.

The scalability curves in Figure 4.8 illustrate that OpenFOAM without DLB exhibits very low efficiency even for increasing problem size. Two major penalties contribute to this: aforementioned parallel communications costs and load imbalance due to dynamic workloads. nSharma addresses the load imbalance penalty in a very effective manner, roughly doubling efficiency for most configurations – the (512,1024K) case of strong scalability can not be taken into account due to the very scarce load per rank. This clearly illustrates that introducing DLB mechanisms results in a very significant reduction of execution time, sustained by an increase in efficiency, i.e. a better utilization of the parallel computing resources.

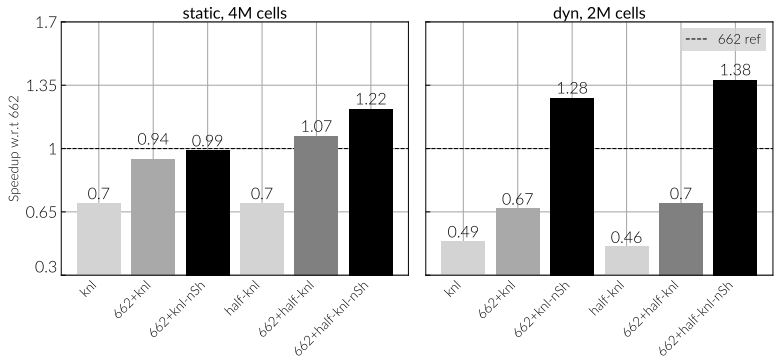


Figure 4.9: Speedup in combining a 662 node and a KNL by using nSharma

4.4.3 Heterogeneity and Dynamic Load Balancing

Effective exploitation of the raw computing capabilities available on heterogeneous systems is hard, with load balancing being one of the main challenges, specially for dynamic workloads.

Figure 4.9 details the performance speedup when combining a KNL node – with two different core configurations, one with the full 64 cores (*knl*) and another with only 32 cores (*half-knl*) – with a 24-core 662 node. Speedup is illustrated w.r.t to the execution time obtained with the node 662 for static (left) and dynamic (right) workloads. By adding a KNL node to a 662 node (*662+knl* and *662+half-knl*) yields no significant performance gain, with a severe deterioration for the dynamic workloads. This is due the imbalance introduced by the large computing power differences between the nodes (as illustrated by the white bars).

By enabling nSharma, the whole system capabilities will be assessed and more load is assigned to 662 node, reducing its idle time and increasing resource utilization. Performance gains between 22% to 38% are observed (**-nSh* bars). The gain is more substantial with dynamic workloads where the potential for load imbalances is larger: heterogeneous resources plus execution time locally varying number of cells. nSharma works at its best under these more challenging conditions, effectively rebalancing the workload and efficiently exploiting the available resources[1].

4.5 Conclusions and Future Work

This contribution proposes and assesses the integration of heterogeneity aware DLB techniques on CFD simulations running on distributed memory heterogeneous parallel clusters (Tier-4 systems). Such simulations most often imply dynamic workloads due to execution time mesh refinement. Combined with

[1] Note that the results indicate that the performance with *half-knl* is higher than using *knl* (full chip). This is due to the lack of optimizations in OpenFOAM targeted for this device. This is thus out of the scope of this thesis and not considered pertinent for this discussion

the hardware heterogeneity such dynamics cause a two-fold load imbalance, which impacts severely on system utilization, and consequently on execution time, if not appropriately catered for. The proposed approach has been implemented as a software package, designated nSharma, which fully integrates with the latest version of OpenFOAM.

Substantial performance gains are demonstrated for both static and dynamic workloads. These gains are shown to be caused by reduced busy times RSD among ranks, i.e. computing resources are kept busy with useful work due to a more effective workload distribution. Strong and weak scalability results further support this conclusion, with nSharma enabled executions exhibiting significantly larger efficiencies for a range of experimental conditions. Performance gains increase with problem size, which is a very desirable feature, since the potential to load imbalances under dynamic loads grows with the number of cells.

Experimental results show that performance gains associated with nSharma are affected by increasing the number of ranks for larger node counts. This is due to inherent increase of load migration costs associated with a growing number of balancing episodes. Future work will necessarily imply addressing this issue, to allow for increased number of parallel resources by further mitigating load migration overheads. Additionally, nSharma will be validated against a more extensive set of case studies and heterogeneous devices; upon successful validation it will be made publicly available in order to foster its adoption by the large community of OpenFOAM users.

Chapter

5 Power Scheduling in Heterogeneous Distributed Systems

Contents

This chapter focuses on the power management challenges of heterogeneous distributed systems. It describes the formulation of an optimization problem that aims at reducing power consumption while minimizing performance degradation, including scenarios with limited power supply. The proposed formulation uses some of the mechanisms described in the previous chapter resulting in a heterogeneity-aware power-adaptive scheduler, integrated into a widely used numerical simulation library. Results are evaluated with multiple configurations and different scenarios.

5.1 Introduction

As discussed in the previous chapter, engineering and scientific computer simulations have become a fundamental tool to analyse complex phenomena and design/verify sophisticated engineering artefacts. Over time both problem size and intended accuracy have increased steadily, resulting in huge workloads which require extended computing capabilities in order to produce results in an appropriate time-frame. Such intensive workloads emphasize the need for larger and powerful parallel supercomputers, further motivating the forthcoming exascale computing era [22], [81]. This shift in computing capabilities poses several challenges, including a **fast-growing power consumption**, with the consequent huge environmental and economic impact. The cost of energy required to power such system will quickly surpass the cost of the physical system itself. Power management becomes of paramount importance, with hardware and, especially, software solutions requiring re-evaluation in terms of power-efficiency to be able to operate under a power limited system.

A common approach to address these power limitations is the adoption of ***hardware overprovisioning***: more parallel computing resources are installed than the organizational power budget allows to operate simultaneously at Thermal Design Power (TDP). TDP is the average maximum power in Watts that the cooling system needs to dissipate – it can also be understood as the average maximum power drawn by a device under any workload. An overprovisioned system requires that operating power limits are enforced by power management software mechanisms which, within a given power budget, will cap the power available to each CU [82], [83]. Limiting power consumption can be accomplished by reducing the CU operating frequency, which obviously impacts its performance. A very simple static strategy is to uniformly cap the power available to every CU, e.g. if the available power budget is s% of the total system TDP, then each node can only use up to s% of its TDP.

However, such uniform and static power allocation strategy can only be optimal if both the available resources and the workload computing requirements are themselves also uniform and static, which is seldom the case. Today's computing systems are rendered heterogeneous in the sense that different generations of hardware coexist in the same system along with a plurality of different devices. This heterogeneity makes the CUs non-uniform with respect to computing capabilities, compromising the

optimality of uniform power allocation.

This is further aggravated by the presence of **dynamic workloads**. In CFD simulations, results' accuracy and relevance are influenced by the discretisation's level of detail, which also determines problem size and thus computational effort. This discretisation can be locally refined as the simulation progresses through time, allowing for higher accuracy in regions of the problem domain where the quantities of interest exhibit more significant local variations; this progressive refinement process generates dynamic workloads and computational requirements will unpredictably vary in runtime among CUs. These imbalances are not accounted for by a static power allocation strategy.

Dynamic workloads executing on heterogeneous parallel systems require dynamic power management mechanisms. By dynamically and at runtime **migrating power** among CUs according to their relative performances and current workload distribution will allow for a more efficient distribution of power, while maximizing performance within a given the power budget. This contribution proposes **RHeAPAS**, which provides power consumption optimization, targeting heterogeneous parallel systems in the context of CFD simulations. This is achieved by integrating power scheduling mechanisms in a widely used CFD application: OpenFOAM.

The proposed runtime is achieved by leveraging the work done in the previous chapter, being deployed as an additional and **innovative functionality of nSharma**. It builds on top of nSharma's resource monitoring and performance model components, as well as on its integration with OpenFOAM. RHeAPAS combines these estimates with a power consumption model based on the CUs' operating frequencies and current workload. A multi-objective minimization problem is then solved: find the frequencies configuration for all CUs that will produce the minimum execution time (maximum performance), while simultaneously using the least amount of total power from the allowed power budget. Integrating RHeAPAS, through nSharma, into current OpenFOAM distributions enables the adoption of power-adaptive scheduling by the CFD community, providing a validated implementation integrated into a widely used scientific and industrial application. Note that this contribution only address and discusses power assignment and performance trade-offs — the workload re-distribution and migration features of nSharma are not applied and are not part of this contribution.

The contributions of this work are summarized as follows:

1. proposal of a power consumption model integrated with a performance model targeting heterogeneous distributed system with dynamic workloads in the context of CFD simulations;
2. proposal of the integration of the power consumption model with the performance model, targeting heterogeneous distributed system with dynamic workloads in the context of CFD simulations. Such integration abides to power budgets while striving for performance optimization;
3. experimental results analysis and validation in multiple parallel heterogeneous system configurations used by the CFD community;

4. deployment of the proposed solutions as an available and free-to-use open-source package, integrated into a widely used scientific and industrial CFD application (OpenFOAM), therefore promoting the adoption of optimized power consumption technologies in CFD simulations.

To the best of author's knowledge, this is the first implementation and integration of power management solutions in OpenFOAM.

5.2 Related Work

Dynamic Voltage and Frequency Scaling (DVFS) mechanisms [38] are commonly used to control CUs' power consumption, such as with **CPU MISER** [84] and **PART** [85], which propose a performance model based on clock cycles per instruction, instructions execution rate and memory accesses to decide on frequency scaling for time intervals, using a user-supplied performance loss parameter to minimize energy utilization. In this work, the user specifies the power-cap and the performance is maximized according to power limitations and system resources.

Jitter [86] performs decisions based on MPI critical path analysis and scales frequency such that all CUs meet at the same point in time. **Adagio** [87] , **Conductor** [88], **GEOPM** [89] also make decisions based on MPI critical paths combined with task models. However, CFD dynamic workloads resulting from cells refinement, render MPI critical path analysis impracticable as results become potentially obsolete across iterations. Additionally, current OpenFOAM nearest neighbour and global communications make task model scheduling unfeasible due to data dependencies. In this work, scaling frequency decisions are made and applied on a per iteration basis given the dynamic nature of the workload.

Nornir [90] and **LEO** [91] target single compute nodes and use machine learning to predict performance and power consumptions for a set of possible configurations; the most appropriate configuration is selected by solving a minimization problem. Dynamic workloads are considered, but high overheads are incurred if the workload varies rapidly relative to the time taken for decision making. The focus is on supporting generic applications, resorting to no specific or dynamic knowledge on the application behaviour. In this work, similar minimization techniques are used, but applied to parallel distributed systems coupled with an application-specific performance and power model; by exploiting previous knowledge on OpenFOAM computation patterns, estimates of the workload near future behaviour incur reduced overhead and attain improved accuracy.

PaViZ [92] proposes a power-adaptive scheduler that distributes a power budget across distributed resources targeting visualization workloads. The associated performance model includes multiple visualization specific details to estimate near future performance. Estimates are then normalized across nodes, resulting in a percentage of the power budget to allocate to each node. This work follows a similar approach by using a performance model targeting CFD workloads, solving a power and execution time

minimization problem constrained by a user-supplied power budget.

Solutions targeting heterogeneous parallel systems are scarce, with some authors addressing heterogeneous single nodes composed of multiple devices (Tier-3). DAG-task scheduling mechanisms [93], [94] are not suited for CFD simulations involving global data dependencies. Tsoi and Luk [95] profile and interpolate performance and power consumption for multiple core and frequency configurations in a CPU+GPU+FPGA node and select one configuration based on a floating point operations per joule metric. Wang and Ren [96] also target single node GPU+CPU using DVFS and iterating through all possible combinations. Liu et al. [97] discuss power-aware analytical models to map multiple applications into a CPU+GPU node and still meet applications' timing requirements, while simultaneously reducing power and energy consumption by applying DVFS techniques.

None of the above approaches accounts for dynamic workloads and none assesses scalable distributed-memory heterogeneous systems or provides validation with large scientific and industrial applications. Additionally, all aim at generic applications resorting to static performance estimates based on generic metrics. The current work addresses these issues and focuses on iterative CFD simulations, using application specific performance and power models, which allow for increased accuracy and reduced overhead performance estimates across different devices.

As described in Section 4.2, there are other libraries available supporting the development of CFD numerical simulations, and the same reasoning is applied as why OpenFOAM was selected as the main target applications. The author sees no reason why conceptual results, such as the power consumption model, and result analysis presented in this document cannot be applied within the context of other similar CFD simulation libraries.

5.3 RHeAPAS

OpenFOAM simulations are organized as solvers, which typically iterate through time in discrete time steps evaluating, at each iteration, the quantities of interest across the cells' mesh that discretizes the problem domain. Each iteration includes multiple inner loops and both local and global communications. With parallel domain decomposition, the initial cells' mesh is decomposed into disjoint subdomains, which are assigned each to a given CU. When applying dynamic mesh control, cells are subdivided and merged according to local variations across iterations. Since the workload is tightly correlated to the number of cells, mesh refinement is the main reason why the workload varies across CUs and from iteration to iteration. This contribution proposes a mechanism to **devise a power schedule** for the next iteration which minimizes that iteration's execution time and power consumption, thus catering for dynamic workloads.

RHeAPAS strongly builds on top of nSharma. It maintains its component-based software architecture in order to seamlessly integrate with OpenFOAM and fully reuses two of its components: the *OPM* and

the *PM*. The former acquires information with respect to raw system behaviour, while the latter uses this data to quantitatively characterize each CU performance. RHeAPAS introduces a new component, the **Power-Adaptive Scheduler (PAS)**, which uses these performance estimates to devise a power schedule, specifying the CUs' frequencies to be set for the next iteration using each CU power API. The following sections summarize the details elaborated in the previous chapter.

5.3.1 Online Profiling Module

This module is responsible for measuring raw timing data by instrumenting OpenFOAM. As discussed in Section 4.3.1, by thoroughly analysing OpenFOAM's code flow a set of low-level routines has been identified, whose execution times strongly correlate with the whole OpenFOAM application timing – this correlation has been empirically verified. This selective profiling approach allows for reduced instrumentation overhead, compared to actually measuring the whole solver code.

A short API has been developed which allows for registering the routines to be measured and to internally classify them as either some form of communication (synchronisation, memory transfer, etc.) or computation. This classification allows measuring and analysing performance individually for each CU, otherwise, execution time would be cluttered by dependencies and communications. These techniques and API should be agnostic to the particular software package being used (other than the identification of the set of representative low-level functions) and can be seamlessly integrated into any other simulation software.

5.3.2 Performance Model

The PM quantifies the performance of each CU used by the application at runtime. This quantification takes into account the specificity of CFD workloads by using a domain-related definition of work unit, upon which workload size can be measured. OpenFOAM is based on Finite Volumes and discretizes the problem domain using the notion of cells. Most OpenFOAM low-level routines exhibit a computational cost proportional to the number of such cells, which are therefore used as the work unit; workload size is quantified as the number of cells.

The performance of each CU p during iteration i is defined as the average time required to process one work unit, denoted by r_p^i, and given by (same as Equation 4.1):

$$r_p^i = \frac{\sum_j^B \beta_p^{j,i}}{N_p^i} \tag{5.1}$$

where N_p^i is the number of cells assigned to CU p and $\beta_p^{j,i}$ is the busy time for each operation j from the set of operations B captured by the OPM. To estimate performance in the near future (e.g. the next iteration) the PM uses a weighted average over a window of previous iterations, denoted as $\tilde{r}_p^i$. This

averaging smooths out outliers and, for dynamic workloads, takes into account different workload sizes as the mesh refinement process refines and merges cells at each CU.

Given the number of work units assigned to each CU, N_p^{i+1}, the PM estimates the execution time for the next iteration, T_p^{i+1}, for all $p \in \{0, 1, ..., P-1\}$, with P being the number of CUs, as given by (Same as Equation 4.2):

$$T_p^{i+1} = \tilde{r}_p^i \times N_p^{i+1} \tag{5.2}$$

Dynamic workloads, resulting from the mesh refinement process, are accounted for by two mechanisms: (i) computing $\tilde{r}_p^i$ as an average over a window of iterations integrates into a single metric potential varying behaviours for different numbers of work units and (ii) re-estimating T_p^{i+1} at the beginning of each iteration allows for a regular accommodation of the new workload characteristics. The system's heterogeneity, on the other hand, is taken into account by measuring and calculating independent $\tilde{r}_p^i$ metrics per CU.

5.3.3 Power-Adaptive Scheduler

An overprovisioned system is characterized by an upper bound on the available power, referred to as the power budget, which has to be distributed across CUs. This power budget is either denoted by τ if expressed in Watts, or denoted by s if expressed as a percentage of the maximum power, i.e., the sum of all the CUs' TDP. A static and uniform power management policy consists of assigning each CU the same percentage s of its TDP. This can be achieved by specifying a maximum capped operating frequency, f_p^{cap}, to each CU p. This power assignment approach can be defined as a power management strategy and it will be referred to as **Uniform Distribution of Power (UDP)**.

As discussed in Section 2.3, power dissipated is correlated with the operating frequency. This **relationship between frequency used and power consumed** can thus be expressed as a function, $\Phi_p(W) = f$, that translates the power assigned to a CU to the corresponding frequency f. Φ_p depends on each CU hardware details and can be modelled in multiple ways, for example, by a tabular function with observed power consumption for each f, or a linear regression based on some observations. The frequency corresponding to a capped power supply, f_p^{cap}, can thus be defined as $f_p^{\text{cap}} = \Phi_p(s \times \text{TDP}_p)$.

Generating a power schedule, i.e. a specification of the power to be used by each CU during the next iteration $i+1$, is formulated as a minimization problem pursuing two objectives: minimization of (i) power usage and (ii) execution time. Let $\mathbf{W}^{i+1}$ be the P-elements vector specifying the operating power for each CU p, W_p^{i+1}, over iteration $i+1$. Clearly, given the power budget τ, it is required that $\sum_{p=0}^{P-1} W_p^{i+1} = \left\|\mathbf{W}^{i+1}\right\|_1 \leq \tau$. In the following a model is developed, inspired in Equation 5.2, to estimate each CU execution time given its allocated power, $T_p^{i+1}(W_p^{i+1})$. Note that due to global synchronisation the iteration execution time is given by Equation 5.3.

$$T^{i+1}(\mathbf{W}^{i+1}) = \max_{p \in \{0,...,P-1\}} T_p^{i+1}(W_p^{i+1}) \tag{5.3}$$

It is well known from reference course books [37] that the time required to process a single cell is given by Equation 5.4,

$$r_p = \frac{\#l_p}{\text{IPC}_p \times f_p} \tag{5.4}$$

where, for each CU p, $\#l_p$ represents the average number of instructions required to compute a cell and IPC_p is the number of instructions per clock cycle[1].

$\#l_p$ and IPC_p are application and CU dependant and approximately constant across all iterations, therefore their ratio can be inferred using known values for r_p and f_p. A set of k initial iterations[2] is computed at f_p^{cap}, which allows the performance model to calculate $\tilde{r}_p^k$ (see Section 5.3). By using $\tilde{r}_p^k$, $\#l_p/IPC_p$ can be approximated by:

$$\frac{\#l_p}{\text{IPC}_p} = \tilde{r}_p^k \times f_p^{\text{cap}} \tag{5.5}$$

The power consumed by a CU can be modelled as described in Equation 2.1 (Section 2.3). Since power dissipated is correlated with the operating frequency, this relationship between frequency used and power consumed can thus be expressed as a function, $\Phi_p(W) = f$, that translates the power assigned to a CU to the corresponding frequency. Φ_p depends on each CU hardware details and can be modelled in multiple ways, for example, by a tabular function with observed power consumption for each f [3], or a linear regression based on some observations.

From Equations 5.2, 5.4, 5.5, and replacing f_p with Φ_p, execution time based on power is given by:

$$T_p^{i+1}(W_p^{i+1}) = \tilde{r}_p^k \times f_p^{\text{cap}} \times \frac{1}{\Phi_p(W_p^{i+1})} \times N_p^{i+1} \tag{5.6}$$

Equation 5.7 presents the minimization problem to be solved (for simplicity index $i+1$ has been omitted), which searches for the schedule $\mathbf{W}$ that yields the minimum execution time and the minimum total power. A scalarization technique, using the coefficients α_1 and α_2, has been applied to combine both objective functions.

$$\begin{aligned} \underset{\mathbf{W}\in\mathbb{R}^P}{\text{minimize}} \quad & \alpha_1 \times T(\mathbf{W}) + \alpha_2 \times \|\mathbf{W}\|_1 \\ \text{subject to} \quad & \|\mathbf{W}\|_1 \leq \tau \\ & T(\mathbf{W}) \leq T(\mathbf{W}^{\text{cap}}) \\ & f_p^{\min} \leq \Phi_p(W_p) \leq f_p^{\max},\ \forall p \in \{0, \ldots, P-1\} \end{aligned} \tag{5.7}$$

The first constraint ensures that the **given power budget limit is not exceed**. The second constraint

[1] For multi-cores CUs, with frequency set per-socket or per-node, $r_p = \#l_p/(c_p \times \text{IPC}_p \times f_p)$, where c_p is the number of cores.
[2] k is parametrized and empirical validation revealed a value of about 15 iterations to be acceptable for most cases.
[3] Typically, frequency is defined in steps ranging from $f_p^{\min}$ to $f_p^{\max}$.

ensures that **estimated execution time is less or equal** than with uniform power capping – minimization could otherwise increase execution time in favour of reduced power consumption. The final constraint ensures that w_p is **within the CU allowed frequency range**. Note that f_p^{max} is the upper limit instead of f_p^{cap}. This is a crucial condition, since it allows selecting large frequencies for CUs with intensive workloads, allowing for higher performance than uniform cap. Each CU power w_p is then mapped to the corresponding operating frequency using Φ_p. In the absence of dynamic workloads, and since compute resources are static along the whole simulation, the schedule is computed only once.

5.4 Results

Experiments use the same ***damBreak*** simulation distributed with OpenFOAM tutorials with the ***interDyMFoam*** solver simulating the multiphase flow of two incompressible fluids – air and water – of a falling block of water. For dynamic workloads, adaptive mesh refinement is applied at each iteration (cells are subdivided into 8 new cells) according to the interface between water and air; cells will thus be refined (and unrefined) following the evolution of the two phases' interface.

The mesh is decomposed using ParMETIS, that creates as many **equally sized partitions** as there are MPI ranks — partitioning is independent of computing capabilities. The assignment is the same as in the previous chapter where each MPI rank is responsible for a computing core. A CU is defined as a compute node composed of multiple cores. Each CU is therefore responsible for a set of partitions, whose numbers of cells can evolve differently for dynamic workloads. Since iteration execution time is proportional to the number of cells and determined by the last rank to finish the iteration, the cell count of the core with the most cells is defined as the number of cells of that CU:

$$N_p^i = \max_{d \in \{0,\ldots,c_p-1\}} N_{p,d}^i \tag{5.8}$$

This aggregation of c_p cores into a single CU facilitates results analysis and avoids using more sophisticated mechanisms requiring per-core frequency scaling. The author sees no reason why the results and discussion provided in this Section cannot be, in general, extended to alternative definitions of CUs.

Experimental results were collected using three configurations from the **SeARCH** cluster (Universidade do Minho, Portugal), as described in Table 5.1. OpenFOAM 2.4.0 was used, compiled with the GNU C Compiler. Frequency scaling is applied per node using the ACPI CPUFreq driver.

The NLopt library [98] is used to solve the minimization problem. Since $T(\mathbf{W})$ (Equation 5.7) is defined using the max function, which is not differentiable, it is replaced by a new decision variable Z and a new constraint: $T_p(W_p) \leq Z, \forall p \in \{0,\ldots,P-1\}$. The objective is moved to the constraints, producing mathematically equivalent results at the cost of some minimization overhead. The coefficients α_1 and α_2 are equally defined as 0.5. $\Phi_p(W)$ (discussed in Section 5.3.3) is approximated using a linear function

Table 5.1: SeARCH Computing nodes and system configurations used in evaluation

Node Tag	Description	f^{min}	f^{max}	TDP
641	2x Ivy Bridge E5-2650v2, 16 cores	1.20GHz	2.60GHz	95W
421	2x Nehalem E5520, 8 cores	1.60GHz	2.27GHz	80W
KNL	Intel Xeon Phi 7210, 64 cores	1.00GHz	1.30GHz	215W
Configuration	**Description**			**Network**
Homogeneous	Multiple 641's			Myrinet
Heterogeneous I	Pair(s) of 641+421			Myrinet
Heterogeneous II	Pair 641+KNL			Ethernet

based on the TDP and corresponding frequency, f_p^{max}, as provided by the manufacturers:

$$\Phi_p(W) = \frac{f_p^{max} \times W}{TDP_p} \tag{5.9}$$

Experimental results are collected for three different levels of power capping, associated with different percentages of TDP: 100%, 85% and 70%. For each of these scenarios, results achieved with a UDP are compared against results obtained with the proposed power-adaptive mechanisms. For the latter, the first 15 iterations are executed at f_p^{cap} to build the performance model.

5.4.1 Performance and Power

Performance gain is defined as the ratio between the execution time obtained with UDP over the execution time obtained with power adaptive scheduling for the entire simulation time span, i.e. how many times the latter is faster than the former.

Used power is always presented as a percentage of the power budget: $\sum_p W_p^i / \tau \times 100$; the reported values are averages over all the iterations, except in Figure 5.2.

Figure 5.1 illustrates the results for multiple heterogeneous configurations with static (first row) and dynamic (second row) workloads. The first two columns use 2 (16 ranks) and 4 nodes (32 ranks), respectively, with Heterogeneous I configuration; the last column uses Heterogeneous II configuration (641+KNL, 80 ranks). The left y-axis shows power used (lower is better), the right y-axis performance gain (higher is better) and the x-axis represents the different power capping levels, s.

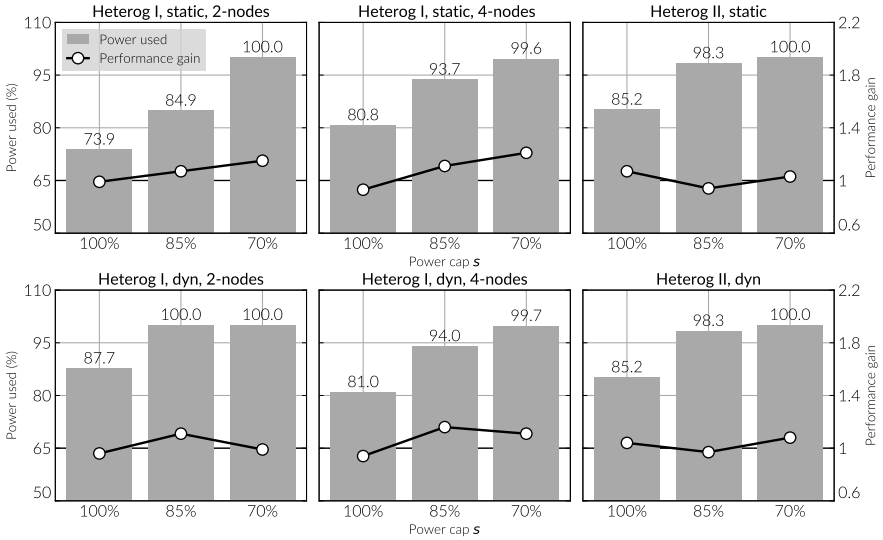


Figure 5.1: Power used and performance gain for (2 and 4 nodes) Heterogeneous I and II with static and dynamic workload in SeARCH. 512K cells for static 256K cells for dynamic.

For the no power limitation case (100%), the power used ranges from 73.9% to 97.2%. For Heterogeneous I with static workload power savings arise from reducing the power assigned to the stronger 641 nodes while assigning enough power to the 421 nodes to prevent performance deterioration. For the dynamic case, less power is assigned to nodes with smaller workloads, properly modulated by the relative performances. For the Heterogeneous I configuration with 2 nodes and dynamic workload, the power used was identical to UDP because the cells assigned to the stronger 641 node were refined, whereas those assigned to the slower 421 nodes sustained much less refinement; remember that refinement occurs along the interface between air and water and the actual assignment to CUs of the cells laying on this interface is not a parameter being controlled and depends on many factors, including the number of nodes in the system. The same reasoning applies to the Heterogeneous II (last column) configuration, where the KNL is slower than the 641, and the latter power is reduced. No performance gain is expected when there is no power cap, since UDP does not limit power usage. The challenge is to attain significant power savings without impacting on performance (performance gain ≈ 1), which has been achieved.

For a cap of 70%, the power budget is significantly reduced, preventing additional power savings. Performance gains ranging from 1.07x to 1.21x are still observed. Power is migrated from stronger (and/or with less workload) to weaker (and/or with more workload) nodes, which run on higher frequencies compared to the UDP, increasing performance. For instance, with two Heterogeneous I nodes and static workload, the UDP limits the power of the slower 421 node to 56W, whereas the proposed model increases it to 78W. The 22W extra are migrated from the faster 641 node which does not require it. Therefore, instead of reducing power usage, the model decides to use all the available power to reduce the unavoidable performance deterioration arising from power capping.

The 85% case is more representative in terms of power budgeting, with the power scheduler balancing

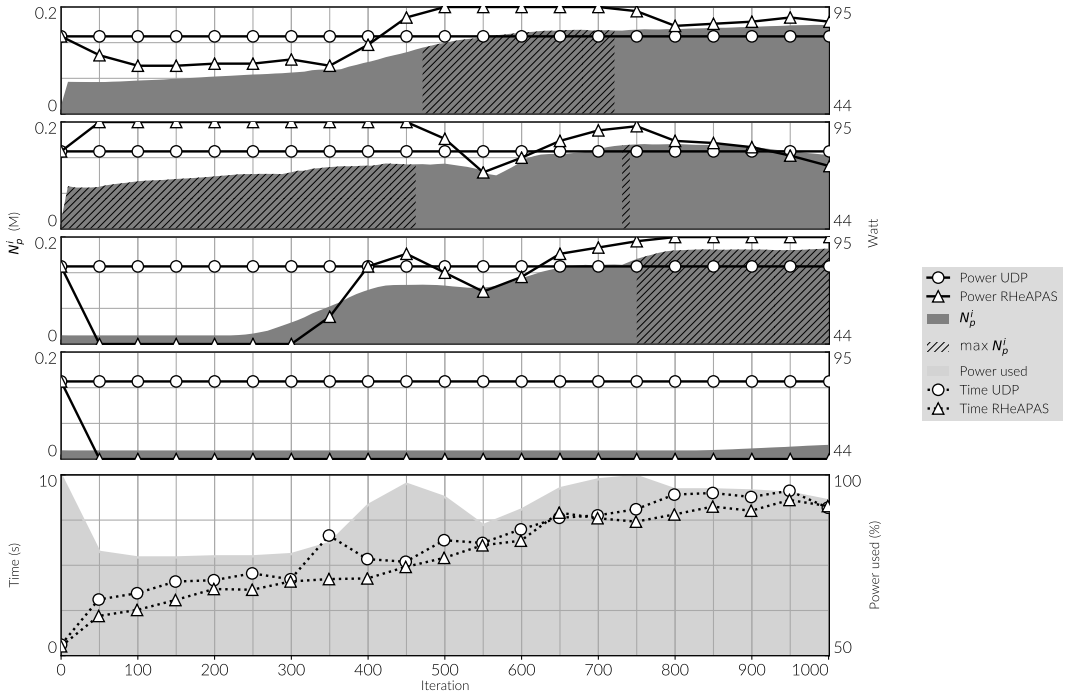


Figure 5.2: Power assignment and iteration execution time along simulation. N_p^i, in the first four rows y-axis, is according to Equation 5.8. 1000 timesteps with 4 homogeneous (641) nodes, dynamic workload and a 85% power budget.

performance gain with power usage reduction. For 4 nodes results show about 90% of power used along with a performance gain of 1.11x (static) and 1.14x (dynamic). For 2 nodes, similar results are observed for the static load. In the dynamic load case, the power used is 100% and performance gain is slightly less, following the same reasoning as for an equal need for power for each node. In general, the model proves to be slightly less effective with Heterogeneous II, due to the short range of frequencies supported by the KNL node – between 1.30 GHz and 1.0 GHz – that significantly reduces the model decision space.

Overall, performance gain increases with the power cap confirming the effectiveness of the model by properly allocating power to where it is most needed under a limited power scenario. The results also reveal that the model is able to successfully reduce power that is wasted by powerful and/or less loaded nodes, particularly when the allowed power budgets are still large.

5.4.2 Dynamic Behaviour

The 4 top rows of Figure 5.2 detail the power assignment per node (4 homogeneous nodes) as time progresses through the simulation (1000 iterations) with dynamic workload and a 85% power budget. The left axis illustrates the number of cells (dark shaded area) per node according to Equation 5.8, and the right axis presents the power assigned to each node at each iteration, ranging from $\Phi_p^{-1}(f_p^{\min})$ to $\Phi_p^{-1}(f_p^{\max})$. The hatched area emphasizes the node with $\max_{p \in \{0,\ldots,P-1\}} N_p^i$ at iteration i. The Figure 5.2 last row shows iteration execution time (left axis) and total power used across all nodes (right axis).

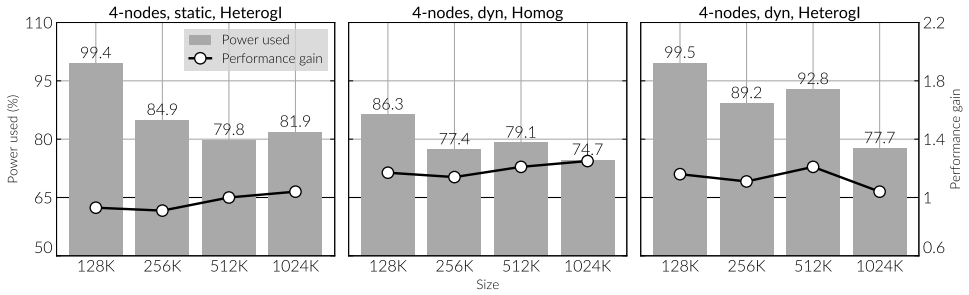


Figure 5.3: Increasing number of cells in the x-axis. 85% power budget, 4 nodes, Heterogeneous I with static load, and Homogeneous I and Heterogeneous I with dynamic workload

The first few iterations use the maximum allowed power to learn the performance model. Significantly more power is then assigned to the second node, which has more cells due to refinement. Between iterations 250 and 400, nodes 1 and 3 get further refined and more power is progressively assigned. Around iteration 500, node 2 reduces the number of cells and power is promptly deallocated with similar behaviour at the end of the simulation. Node 4 has the least number of cells across the whole simulation so minimum power is assigned.

Roughly between iteration 50 and 450, node 2 has the maximum number of cells which thus dictates execution time (note that the dashed area indicates the node with maximum cells). In this segment, the power assigned to this goes above the power assigned to UDP (81W) which results in the performance gain illustrated in the last row. The same behaviour is observed for the rest of the simulation with different nodes. The power scheduler ability to adapt to variable and unpredictable workloads is clearly illustrated, as well as how the model is able to extract performance gains with effective and educated power budget distribution.

5.4.3 Scaling Problem Size and Resources

Figure 5.3 shows the results for an 85% power budget with increasing problem size (number of cells, x-axis) for 4 nodes, Heterogeneous I with static load, and Homogeneous I and Heterogeneous I with dynamic workload. The results are fairly consistent across configurations and reveal an increasing reduction in power used as well a minor increase in performance gain as workload increases. In-depth analysis of the results revealed that, for the static load and heterogeneous case, as the problem size increases so does the computation to communication ratio; the performance model becomes more accurate resulting on a more effective power schedule.

For dynamic load and heterogeneous configuration, only two of the nodes performed refinement, with one of them sustaining more cells than the other. This gap between these two nodes increased substantially with the number of initial cells – from 10% for 128K to 150% for 1024K. This results in significantly less power assigned to the node with fewer cells as the number of initial cells increases. The last plot

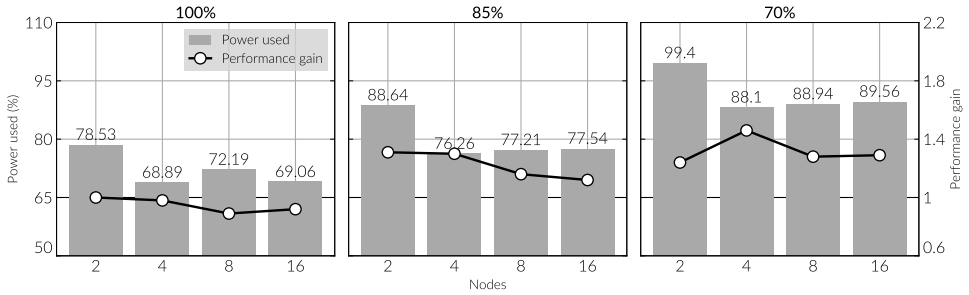


Figure 5.4: Weak scaling based analysis, homogeneous nodes increasing in the x-axis. 512K, 1024K, 2048K and 4096K as number of cells respectively, and dynamic workload.

shows the combination of dynamic load with heterogeneous configuration where results vary significantly as the number of cells increase. As the initial number of cells changes, the nodes responsible to perform refinement also changed resulting in different power scheduling decisions considering different node performances – for 128K and 512K the two 641 performed refinement whereas for 256K and 1024K, a 641 and a 421 performed refinement. These results demonstrate that the proposed mechanisms sustain an increased effectiveness with large workloads, which is a fundamental result in the context of CFD simulations.

Figure 5.4 illustrates weak scalability analysis with the number of cells (512K, 1024K, 2048K and 4096K) increasing linearly with the number of nodes (2 (8 ranks) to 16 (64 ranks)). Results show a slight reduction in power used as nodes are added, especially for the 100% case. This is because adding more nodes will increase the number of nodes that can have their power reduced without affecting performance (performance is dictated by the node with more cells), therefore leading to a reduction in required power. Similar reasoning can be applied to the performance gain. Performance gain results from nodes with more workload running with more power compared to UDP. Once the maximum power is assigned to these nodes, adding more nodes will have no effect, resulting in the same performance. In fact, a reduction in performance gain is observed as a consequence of the severe impact in the computation to communication ratio due to the increased number of ranks (added workload is not enough to compensate the increased communication overhead). Nevertheless, performance improvements range from 1.06x to 1.41x.

5.4.4 Energy Saved

Power and execution time are two fundamental components that directly contribute to minimize the energy consumption. By reducing the time required to execute the application and thus the time during which energy is being consumed, and/or by reducing the power delivery rate, yields a combined reduction of energy consumed. Total energy consumed, E, is calculated based on the sum of the energy consumed by each individual iteration: $E = \sum_i \left(\sum_p W_p^i \times T^i \right)$; where T^i is iteration's i execution time. E was evaluated for UDP (E_{UDP}) and RHeAPAS (E_{RHeAPAS}), and their differences normalized, resulting in the percentage of

energy saved (E_{saved}) by using RHeAPAS: $E_{saved} = (E_{UDP} - E_{RHeAPAS}) / E_{UDP} \times 100$.

Figure 5.5 shows E_{saved}; the first two rows illustrate the heterogeneous configurations (same as in Figure 5.1). For static loads and no power limit, energy savings over 25% are observed, essentially due to the reduced power usage. Used power increases with total available power, resulting in less energy saved for higher power availability. The third row illustrates increasing problem size (same as in Figure 5.3), clearly demonstrating higher energy savings for larger problems. The last row illustrates the increasing number of nodes (as in Figure 5.4) where consistent energy savings of around 30% are observed for the three power limits. Overall, the proposed mechanisms prove to be substantially more effective than a UDP in multiple scenarios and attending to different power caps.

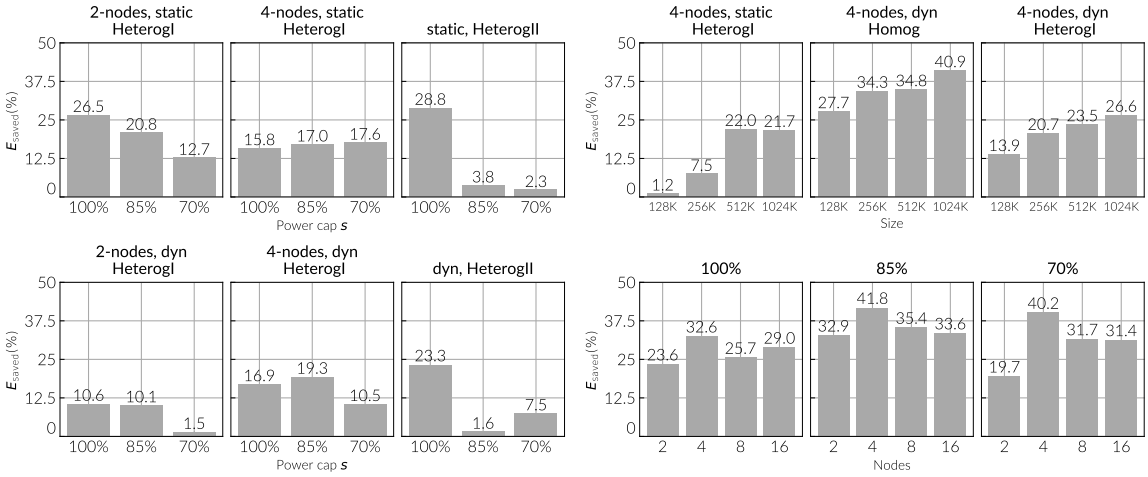


Figure 5.5: Energy consumption reduced for the same configurations of the previous plots. In the first two rows, 512K cells for static 256K cells for dynamic. 85% limit of power for the third row and the last row is a weak scaling with homogeneous 641 nodes with increasing cells (256K, 512K, 1024K and 2048K) and dynamic workload.

5.5 Conclusions and Future Work

This contribution proposes and assesses a runtime power scheduler, which optimizes power consumption for overprovisioned heterogeneous clusters, in the context of CFD simulations. Such simulations often imply dynamic workloads due to execution time mesh refinement that combined with hardware heterogeneity, result in non-optimal power consumption and/or performance degradation when a power supply limit is applied. The proposed approach combines power usage reduction with execution time minimization by formulating an optimization problem that devises a power schedule satisfying both objectives while attending to a power limit. The proposed approach has been implemented as an additional software component of nSharma, which fully integrates with OpenFOAM.

Results, in general, show a substantial reduction in power used for static and dynamic workloads with no performance deterioration. When the power budget is significantly reduced, performance improvements

are observed when compared to a uniform distribution of power. These gains are shown to be the result of adaptively assigning the power to where it is most needed. Power from faster nodes and/or nodes with less workload is migrated to slower and busier nodes, resulting in an overall reduction in power used and performance gain. Assessments with multiple problem sizes are also included, revealing an increased effectiveness as problem size increases. Increasing number of compute units are also evaluated, demonstrating a consistent reduction in power used along with performance improvements, however, the latter slightly affected by the computation to communication ratio. Since power supply and performance basically define the energy consumption of an application, an energy saved analysis reveals that a substantial reduction in energy is observed, in many cases over 30%.

Experimental results show that the effectiveness of the proposed model is, in some cases, affected by the range of frequencies available for each compute unit. Future work will account for available ranges, producing better results when compute units with a short range of frequencies is present. Additionally, the runtime will be validated against a more extensive set of case studies and heterogeneous computing units; upon successful validation, RHeAPAS will be made publicly available to be integrated into clusters running OpenFOAM.

Chapter

6 Conclusions and Future work

Contents

The best thing about the future is that it comes one day at a time.

Abraham Lincoln

In this final chapter, the general conclusions achieved in the thesis are identified. The three contributions are discussed altogether and related to this thesis hypothesis. The future work is also discussed, with a new model being proposed for development and assessment.

6.1 Conclusions

This thesis addresses the heterogeneous nature of today's parallel computing systems in the context of numerical computer simulations with focus on dynamic workloads. It approaches the multiple challenges that these systems pose, particularly when computing dynamic and irregular workloads originated from large and complex numerical simulations such as CFD simulations with adaptive mesh refinement. The challenges include performance imbalances originated from nodes or devices with different computing capabilities, performance non-portability, disjoint memory address spaces, non-portable code and emerging challenges related to power management. These challenges are further aggravated with the dynamic and unpredictable nature of the workload since it produces an arbitrary amount of computational effort as well as code divergence and branching workflow that current computing software solutions and paradigms do not address.

The heterogeneous challenges posed by current computing systems together with dynamic workloads form a *Two-fold Challenge* that this thesis proposes to address using a combination of mechanisms that are designed, implemented and validated across a conceptual 4-Tier parallel hierarchy defined in this document. These mechanisms include a unified execution and programming model, transparent data management systems, heterogeneity aware dynamic load balancing and heterogeneity aware power management. This thesis hypothesis is thus that these techniques can be used to face the multiple challenges raised across the 4 tiers of parallelism in order to increase the development productivity, compute efficiency and a proper balance between performance extracted and power used.

The contributions of this thesis were organized in three main parts:

Heterogeneity challenges in Tier-1,2 and 3 parallel systems In this contribution, single node multi-device (Tier-3) systems were addressed and a unified task-based programming and execution model tailored to efficiently execute data-parallel regular and irregular applications was proposed. Among other mechanisms, the execution model includes the integration of persistent kernels combined with a tailored API allowing users to express irregular applications towards increasing the performance extracted from the Tier-2 and 1 parallel levels. Results reveal a gain of up to 84% in some applications along with consistent levels of parallel efficiency as resources are added.

Heterogeneity challenges in Tier-4 parallel systems In this contribution the challenges with multi-node distributed memory systems (Tier-4) were addressed and the proposed approach is directly integrated and evaluated with a widely used CFD library (OpenFOAM). The contribution evaluated

the combination of a DLB system with an application-oriented performance model as a mean to increase resource utilization in performance and workload imbalanced systems. Speed-ups larger than 2 were achieved with some configurations and an increased parallel efficiency when compared with the out-of-the-box simulation time results.

Power-management in Tier-4 heterogeneous systems The fast-growing power consumption was addressed in this contribution by devising and solving an optimization problem in order to improve power efficiency and performance in power-limited scenarios. The proposed model is formulated based on two objectives: power consumption minimization and performance maximization. Heterogeneity awareness is provided by a performance model and power assignment decisions are adaptively performed at runtime. Reductions in power consumption over 25% were observed in some configurations with fairly acceptable adaptivity to dynamic workloads and resources variability. Gains over 40% in energy are also observed in some configurations.

These contributions show that unified programming and execution models are an effective way of increasing productivity and performance by hiding the main hurdles that heterogeneous parallel systems pose to applied science experts. They provide a mechanism for transparent handling of multiple architectures with different performance levels offered by different computing units. These unified runtime systems must provide a data-management system in order to further enhance productivity and also increase scheduling opportunities to push forward performance boundaries. Special focus was set on scheduling and how persistent kernels may be explored in order to increase the performance of irregular applications in highly parallel architectures. Results revealed substantial gain when using these tools as long as the application sustains enough computational effort to mitigate the workload management overheads.

Results also show that Dynamic Load Balancing (DLB) techniques are capable of substantially increasing the performance of a complex state-of-the-art CFD software package in heterogeneous distributed memory systems. By resorting to a thorough combination and design of a profiling mechanism, a tailored performance model, a decision module and a repartitioning module, a runtime system can be integrated into a numerical simulation package allowing it to effectively account for the differences in performance across nodes including particularly challenging scenarios like dynamic workload simulations.

Finally, this document describes a formulation of an optimization problem that distributes a power budget and tries to minimize power consumption while also minimizing the performance penalty. It explores some of the modules used in DLB techniques, such as the tailored performance model, and the results show that power consumption can be effectively reduced without affecting performance. The model also considers limited power supply scenarios which allows for the model to increase the performance when compared to simpler power limitation approaches. This further increases the benefits of using such optimization formulations in heterogeneous systems resulting in not only power consumption reduction but also a reduction in energy used as a consequence of shorter execution times.

It is thus the author belief that the results presented in this document validate the hypothesis put forward by this research work. The mechanisms identified were able to effectively address the multiple challenges that parallel heterogeneous systems pose, in particular, they were able to address the two-fold challenge defined by combining these challenges with dynamic workloads. The next section discusses the future work and it proposes an approach to extend the optimization problem used in RHeAPAS (Section 5.7) in order to include dynamic load balancing.

6.2 Future Work

In general, the computing platforms used in experimentation throughout the thesis may be classified as small-medium sized systems. Although the author believes that the variety of systems used are the minimum required to validate the proposed goals of this thesis, larger and more heterogeneous systems should be tested. This includes validation with hundreds of nodes as well as systems with a larger number of different devices (higher heterogeneity levels). Larger scale systems potentially introduce other challenges (e.g. higher communication overheads) that need to be accounted for in the proposed mechanisms. Analysing other different computing devices will also provide a better insight in how the models can be further developed to increase the support for arbitrary heterogeneity. For instance, RHeAPAS could benefit from a generic mechanism that would account for the different frequency steps that each individual device potentially has.

Specific to the contributions described in Chapter 4 and 5, the current prototype implementation requires some changes and subsequent re-compilation of the OpenFOAM solvers. This can be completely removed by integrating the required changes in the OpenFOAM core libraries, omitting thus any changes to the solvers required to the OpenFOAM programmers. An extremely limiting feature in OpenFOAM is the requirement of having at least one cell assigned to each instanced rank (no zero-sized partitions). This inhibits nSharma to simply deactivate a CU that is too slow to have any benefit in assigning any work to it. Enabling this in OpenFOAM would significantly increase the benefit of using the proposed mechanisms.

It is crucial that the physical simulation results are not affected by nSharma or RHeAPAS, therefore, the simulation results achieved (e.g. velocity, pressure, etc) need to be properly validated. This requires the implementation of a thorough methodology to compare the results achieved with the proposed mechanisms against the out-the-box simulation values. It is also required to further validate the proposed mechanisms with more simulations cases as well as different OpenFOAM solvers. Since the mesh repartitioning and cell migration is related with the mesh geometry and cell distribution in space, it is required to further assess the behaviour of nSharma with different meshes and simulation workflows.

Regarding the optimization problem used in RHeAPAS, note that the coefficients α_1 and α_2 are equally defined as 0.5 in the experimental tests. This means that the same weight is given to both objectives: per-

formance and power consumption. Further tests with different values for these coefficients can be made in order to assess their impact on the results. Finally, the contributions in Chapter 5 are useful essentially to system administrators and managers. An interface is thus required that connects the tools used by these administrators to the mechanisms proposed in order to be transparently used and parametrized according to system characteristics.

6.2.1 Combining Power Management with Load Balancing

In Chapter 5 a formulation of an optimization problem is proposed in order to find the best trade-off between power and performance that would minimize the power used. Solving the optimization problem consists on finding $\mathbf{W}$ (power assigned to each CU) that minimizes two combined functions, the estimated execution time $T(\mathbf{W})$ and the total power used $\|\mathbf{W}\|_1$ (Equation 5.7). Since no dynamic load balancing is used in this contribution, the number of cells assigned to each CU is known at the start of each iteration and it is used in Equation 5.6 to estimate the execution time with a provided W_p.

This approach can be extended by adding a new set of unknowns to the minimization problem that represent the number of cells N_p^{i+1} assigned to each CU p, i.e., allowing for cell migration among CUs. The N_p^{i+1} in Equation 5.6 can parametrized resulting in:

$$T_p^{i+1}(W_p^{i+1}, N_p^{i+1}) = \tilde{r}_p^k \times f_p^{\text{cap}} \times \frac{1}{\Phi_p(W_p^{i+1})} \times N_p^{i+1} \tag{6.1}$$

Subsequently, Equation 5.3 becomes:

$$T^{i+1}(\mathbf{W}^{i+1}, \mathbf{N}^{i+1}) = \max_{p \in \{0,\ldots,P-1\}} T_p^{i+1}(W_p^{i+1}, N_p^{i+1}) \tag{6.2}$$

And finally, by adding the new set of unknowns to the optimization problem the following is achieved (for simplicity index $i+1$ has been omitted):

$$\begin{aligned}
\underset{\mathbf{W}\in\mathbb{R}^P, \mathbf{N}\in\mathbb{N}^P}{\text{minimize}} \quad & \alpha_1 \times T(\mathbf{W}, \mathbf{N}) + \alpha_2 \times \|\mathbf{W}\|_1 \\
\text{subject to} \quad & \|\mathbf{W}\|_1 \leq \tau \\
& \|\mathbf{N}\|_1 = N_{\text{total}} \\
& \|\mathbf{N}\|_0 = P \\
& T(\mathbf{W}, \mathbf{N}) \leq T(\mathbf{W}^{\text{cap}}, \mathbf{N}) \\
& f_p^{\min} \leq \Phi_p(W_p) \leq f_p^{\max}, \ \forall p \in \{0, \ldots, P-1\}
\end{aligned} \tag{6.3}$$

Note that the constraints $\|\mathbf{N}\|_1 = N_{\text{total}}$ and $\|\mathbf{N}\|_0 = P$ have been added in order to ensure that all the cells are assigned and no zero-sized domains are created, respectively.

Solving this problem will thus search for the best $\mathbf{W}$ and $\mathbf{N}$ that combined will minimize execution time and

power consumption. Note that $\tau(\mathbf{W}, \mathbf{N})$ is based on the performance model, therefore the performance of each CU will also be a defining factor on how to devise $\mathbf{N}$, which is the main purpose of a dynamic load balancing mechanism. The decisions of this model are significantly different from the previous model. The system has now the ability to migrate cells[1] among CUs, balancing between assigning cells to the fastest CUs and the ones that consume the least power. This significantly increases the depth of the decisions made and potentially resulting in substantial gains in energy consumption compared to the ones achieved in Chapter 5.

Note that this approach was not implemented in any way, the optimization model was formulated and presented here, but no validation or assessment was performed. To implement this new approach, most of the mechanisms in Chapter 4 are required. In fact, solving this problem can replace the nSharma linear system of equations described in Section 4.3.3, Equation 4.4. This will produce the N_p^{i+1} values that can then be used in the following pipelined components and finalize with the assignment of the frequencies. Note that, by adding $\mathbf{N}$ as a new set of variables to search, the complexity of solving the model is potentially higher as there are significantly more possible combinations. The overhead and cost of solving such system at runtime must be re-assessed.

A particular advantage of this model is that it provides the ability to, according to the system circumstances, dynamically configure the application to leverage either less power consumption or greater performance. For instance, an administration entity (either an automatic system or human system administrator) can parametrize the execution of the application to leverage power consumption reduction if the system is under high load. The model can thus migrate the work to the most suitable CUs that would minimize performance degradation and define the most suitable frequencies to achieve such goal. On the other hand, if the system has all the available power to compute the application, the model may find the best cell distribution to achieve the best performance, disregarding any concerns with power consumption.

[1] implicitly, by devising the number of cells assigned

Bibliography

[1] J. Dongarra, “Trends in high performance computing: a historical overview and examination of future developments,” *IEEE Circuits and Devices Magazine*, Jan. 2006. DOI: 10.1109/MCD.2006.1598076.

[2] L. Eeckhout, “Heterogeneity in response to the power wall,” *IEEE Micro*, 2015. DOI: 10.1109/MM.2015.86.

[3] A. Ghuloum, “Face the inevitable, embrace parallelism,” *Communications of the ACM*, Sep. 2009. DOI: 10.1145/1562164.1562179.

[4] M. Zahran, “Heterogeneous computing: here to stay,” *Commun. ACM*, 2017. DOI: 10.1145/3024918.

[5] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, “The impact of performance asymmetry in emerging multicore architectures,” *ACM SIGARCH Computer Architecture*, May 2005. DOI: 10.1145/1080695.1070012.

[6] J. Dongarra, P. Beckman, T. Moore, *et al.*, “The international exascale software project roadmap,” *International Journal of High Performance Computing Applications*, 2011. DOI: 10.1177/1094342010391989.

[7] V. W. Lee, P. Hammarlund, R. Singhal, *et al.*, “Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu,” *ACM SIGARCH Computer Architecture News*, Jun. 2010. DOI: 10.1145/1816038.1816021.

[8] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, “Data-aware task scheduling on multi-accelerator based platforms,” in *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, IEEE, Dec. 2010, ISBN: 978-1-4244-9727-0. DOI: 10.1109/ICPADS.2010.129.

[9] H. K. Versteeg and W. Malalasekera, *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*, Second. Pearson Education, 2007, ISBN: 978-0131274983.

[10] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on gpus,” in *Proceedings - 2012 IEEE International Symposium on Workload Characterization, IISWC 2012*, 2012, ISBN: 9781457720642. DOI: 10.1109/IISWC.2012.6402918.

[11] P. Colella, J. Bell, N. Keen, T. Ligocki, M. Lijewski, and B. V. Straalen, “Performance and scaling of locally-structured grid methods for partial differential equations,” *Journal of Physics: Conference Series*, Jul. 2007. DOI: 10.1088/1742-6596/78/1/012013.

[12] M. Burtscher and K. Pingali, “An efficient cuda implementation of the tree-based barnes hut n-body algorithm,” in *GPU Computing Gems Emerald Edition*, Elsevier, 2011, ISBN: 0123849888. DOI: 10.1016/B978-0-12-384988-5.00006-1.

[13] I. Sadeghi, B. Chen, and H. W. Jensen, "Coherent path tracing,"
Journal of Graphics GPU Game Tools, 2009. DOI: 10.1080/2151237X.2009.10129279.

[14] M. Kalos and P. Whitlock, *Monte Carlo methods*, Second. 2008, ISBN: 978-3527407606.

[15] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," *ACM Transactions on Graphics*, 2003. DOI: 10.1145/882262.882364.

[16] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of delaunay and voronoi diagrams," *Algorithmica*, Jun. 1992. DOI: 10.1007/BF01758770.

[17] J. Barnes and P. Hut, "A hierarchical o(n log n) force-calculation algorithm," *Nature*, Dec. 1986. DOI: 10.1038/324446a0.

[18] H. P. Zhu, Z. Y. Zhou, R. Y. Yang, and A. B. Yu, "Discrete particle simulation of particulate systems: a review of major applications and findings," *Chemical Engineering Science*, Dec. 2008. DOI: 10.1016/j.ces.2008.08.006.

[19] R. Lhner, *Applied Computational Fluid Dynamics Techniques*.
Chichester, UK: John Wiley and Sons, Ltd, Mar. 2008, ISBN: 9780470989746.

[20] R. Löhner,
Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods.
2008, ISBN: 9780470989746. DOI: 10.1002/9780470989746.

[21] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges,"
Springer, Berlin, Heidelberg, 2011, ISBN: 978-3-642-19327-9, 978-3-642-19328-6.
DOI: 10.1007/978-3-642-19328-6_1.

[22] U. D. of Energy, "2013 exascale operating and runtime systems," Tech. Rep., 2013.
[Online]. Available: https://science.energy.gov/%7B~%7D/media/grants/pdf/lab-announcements/2013/LAB%7B%5C_%7D13-02.pdf.

[23] Top500, *Top500 supercomputer site*, 2018.
[Online]. Available: http://www.top500.org/site/48958.

[24] Nvidia, *Cuda programming guide*, 2018. [Online]. Available:
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[25] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*. 2003,
ISBN: 0201648652.

[26] MPI-Forum, *Mpi: a message-passing interface standard*, 2015.
[Online]. Available: https://www.mpi-forum.org/.

[27] OpenMP Architecture Review Board, *Openmp api specification*, 2015.
[Online]. Available: https://www.openmp.org/specifications/.

[28] Intel, *Intel threading building blocks developer reference*, 2018.
[Online]. Available: https://www.threadingbuildingblocks.org/documentation.

[29] ——, *Intel cilk plus reference manual*, 2016.
[Online]. Available: https://www.cilkplus.org/cilk-documentation-full.

[30] Microsoft, *.net framework api reference*, 2017.
[Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/.

[31] IEEE, *1003.1-2017 - ieee standard for information technology–portable operating system interface (posix(r)) base specifications*, 2017.
[Online]. Available: https://ieeexplore.ieee.org/document/8277153/?denied.

[32] *Boost c++ library documentation*, 2018. [Online]. Available: https://www.boost.org/doc.

[33] *Iso/iec 14882:2011 information technology – programming languages – c++*, 2011.
[Online]. Available: https://www.iso.org/standard/50372.html.

[34] J. Diaz, C. Munoz-Caro, A. Nino, C. Muñoz-Caro, and A. Niño, "A survey of parallel programming models and tools in the multi and many-core era,"
IEEE Transactions on Parallel and Distributed Systems, 2012. DOI: 10.1109/TPDS.2011.308.

[35] Khronos, *The opencl specification*, 2018.
[Online]. Available: https://www.khronos.org/opencl/.

[36] *Openacc programming and best practices guide*, 2015.
[Online]. Available: https://www.openacc.org/resources.

[37] D. A. Patterson and J. L. Hennessy,
Computer Organization and Design, Fifth Edition: The Hardware/Software Interface.
Morgan Kaufmann, 2014, ISBN: 9780124077263.

[38] G. Valentini, W. Lassonde, S. Khan, *et al.*, "An overview of energy efficiency techniques in cluster computing systems," *Cluster Computing*, 2013. DOI: 10.1007/s10586-011-0171-x.

[39] *Advanced configuration and power interface specification revision 5.0a*, 2013.
[Online]. Available: http://www.acpi.info/spec.htm.

[40] J. Feo, O. Villa, A. Tumeo, and S. Secchi, "Irregular applications: architectures and algorithms,"
in *Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms*,
ser. IAAA '11, 2011, ISBN: 9781450311212. DOI: 10.1145/2089142.2089144.

[41] P. Glasserman, *Monte Carlo Methods in Financial Engineering*. 2003, ISBN: 978-0-387-21617-1.

[42] M. Pedersen, "Monte carlo simulation in financial valuation," Hvass Laboratories, Tech. Rep., 2014.
DOI: 10.2139/ssrn.2332539.

[43] M. Pharr and G. Humphreys, *Physically based rendering: from theory to implementation*, 2nd.
Morgan Kaufmann, 2010, ISBN: 978-0123750792.

[44] C. Zhu and Q. Liu, “Review of monte carlo modeling of light transport in tissues,” *Journal of Biomedical Optics*, 2013. DOI: 10.1117/1.JBO.18.5.050902.

[45] R. Dolbeau, S. Bihan, and F. Bodin, “Hmpp: a hybrid multi-core parallel programming environment,” in *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[46] G. Diamos and S. Yalamanchili, “Harmony: an execution model and runtime for heterogeneous many core systems,” in *Proceedings of the 17th international symposium on High performance distributed computing*, Jun. 2008. DOI: 10.1145/1383422.1383447.

[47] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, “Merge: a programming model for heterogeneous multi-core systems,” in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIII, 2008. DOI: 10.1145/1346281.1346318.

[48] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, “Xkaapi: a runtime system for data-flow task programming on heterogeneous architectures,” in *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, IEEE, May 2013, ISBN: 978-1-4673-6066-1. DOI: 10.1109/IPDPS.2013.66.

[49] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: expressing locality and independence with logical regions,” in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012, ISBN: 9781467308069. DOI: 10.1109/SC.2012.71.

[50] C.-K. Luk, S. Hong, and H. Kim, “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping,” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009. DOI: 10.1145/1669112.1669121.

[51] J. A. Pienaar, A. Raghunathan, and S. Chakradhar, “Mdr: performance model driven runtime for heterogeneous parallel platforms,” in *Proceedings of the international conference on Supercomputing*, May 2011. DOI: 10.1145/1995896.1995933.

[52] C. Augonnet, S. Thibault, R. Namyst, and P.-a. Wacrenier, “Starpu : a unified platform for task scheduling on heterogeneous multicore architectures,” in *Euro-Par 2009 Parallel Processing 15th International Euro-Par Conference*, 2009. DOI: 10.1002/cpe.1631.

[53] H. Topcuoglu and S. Hariri, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, Mar. 2002. DOI: 10.1109/71.993206.

[54] R. Ribeiro, "Portability and performance in heterogeneous many-core systems," MSc thesis, University of Minho, 2011. [Online]. Available: http://hdl.handle.net/1822/28170.

[55] D. Cederman and P. Tsigas, "On dynamic load balancing on graphics processors," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware*, ser. GH '08, 2008. DOI: 10.2312/EGGH/EGGH08/057–064.

[56] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the gpu," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10, 2010. DOI: 10.2312/EGGH/HPG10/029–037.

[57] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the High-Performance Graphics 2009*, 2009. DOI: 10.1145/1572769.1572792.

[58] M. Steinberger, B. Kainz, B. Kerbl, *et al.*, "Softshell : dynamic scheduling on gpus," *Journal ACM Transactions on Graphics*, 2012. DOI: 10.1145/2366145.2366180.

[59] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, Aug. 2007. DOI: 10.1177/1094342007078442.

[60] NVIDIA, *Nvidia cuda basic linear algebra subroutines (cublas)*, 2018. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html.

[61] Intel, *Developer reference for intel math kernel library 2018 - c*, 2018. [Online]. Available: https://software.intel.com/en-us/mkl-developer-reference-c.

[62] *Luxcorerender*, 2018. [Online]. Available: https://luxcorerender.org/.

[63] R. Chamberlain, D. Chace, and A. Patil, "How are we doing? an efficiency measure for shared , heterogeneous systems," in *ISCA 11th, International Conference on Parallel and Distributed Computing Systems*, 1998.

[64] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Sci. Program.*, 2010. DOI: 10.1155/2010/540159.

[65] G. Da Costa, T. Fahringer, J.-A. Rico-Gallego, *et al.*, "Exascale machines require new programming paradigms and runtimes," *Supercomputing Frontiers and Innovations*, 2015. DOI: 10.14529/jsfi150201.

[66] OpenFOAM Foundation, *Openfoam users' guide*, 2018.
[Online]. Available: https://cfd.direct/openfoam/user-guide/.

[67] *Ansys fluent*, 2018.
[Online]. Available: https://www.ansys.com/products/fluids/ansys-fluent.

[68] *Ansys cfx*, 2018.
[Online]. Available: https://www.ansys.com/products/fluids/ansys-cfx.

[69] *Cd-adapco star-ccm+*, 20178.
[Online]. Available: https://mdx.plm.automation.siemens.com/star-ccm-plus.

[70] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel diffusion schemes for repartitioning of adaptive meshes," *Journal of Parallel and Distributed Computing*, 1997.
DOI: 10.1006/jpdc.1997.1410.

[71] C. Chevalier and F. Pellegrini, "Pt-scotch: a tool for efficient parallel graph ordering," *Parallel Computing*, 2008. DOI: 10.1016/j.parco.2007.12.001.

[72] A. Basermann, J. Clinckemaillie, T. Coupez, *et al.*, "Dynamic load-balancing of finite element applications with the drama library," *Applied Mathematical Modelling*, 2000.
DOI: 10.1016/S0307-904X(00)00043-3.

[73] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali, "A load balancing framework for adaptive and asynchronous applications,"
IEEE Transactions on Parallel and Distributed Systems, 2004. DOI: 10.1109/TPDS.2004.1264800.

[74] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan, "Design of dynamic load-balancing tools for parallel applications,"
Proceedings of the 14th international conference on Supercomputing - ICS '00, 2000.
DOI: 10.1145/335231.335242.

[75] J. Faik, J. E. Flaherty, L. G. Gervasio, J. D. Teresco, and K. D. Devine,
"A model for resource-aware load balancing on heterogeneous clusters,"
Williams College Department of Computer Science, Tech. Rep., 2005. [Online]. Available:
http://j.teresco.org/research/publications/tpds05/tpds05.pdf.

[76] J. A. Martínez, E. M. Garzón, A. Plaza, and I. García, "Automatic tuning of iterative computation on heterogeneous multiprocessors with adithe," *Journal of Supercomputing*, 2011.
DOI: 10.1007/s11227-009-0350-1.

[77] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on highly heterogeneous hpc platforms," *Parallel Processing Letters*, 2011.
DOI: 10.1142/S0129626411000163.

[78] D. Clarke, Z. Zhong, V. Rychkov, and A. Lastovetsky, “Fupermod: a software tool for the optimization of data-parallel applications on heterogeneous platforms,” *The Journal of Supercomputing*, 2014. DOI: 10.1007/s11227-014-1207-9.

[79] Z. Zhong, “Optimization of data-parallel scientific applications on highly heterogeneous modern hpc platforms,” PhD thesis, 2014.

[80] K. Mooney and J. Papper, “Implementation of a moving immersed boundary method on a dynamically refining mesh with automatic load balancing,” in *10th OpenFOAM Workshop*, 2015.

[81] Europe, “Cooperation framework on high performance computing,” Tech. Rep., 2017. [Online]. Available: https://ec.europa.eu/digital-single-market/en/news/eu-ministers-commit-digitising-europe-high-performance-computing-power.

[82] Z. Zhang, M. Lang, S. Pakin, and S. Fu, “Trapped capacity: scheduling under a power cap to maximize machine-room throughput,” in *Proceedings of E2SC 2014: 2nd International Workshop on Energy Efficient Supercomputing - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, ISBN: 9781479970360. DOI: 10.1109/E2SC.2014.10.

[83] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, “Exploring hardware overprovisioning in power-constrained, high performance computing,” in *Proceedings of the 27th international ACM conference on International conference on supercomputing - ICS ’13*, New York, New York, USA: ACM Press, 2013, ISBN: 9781450321303. DOI: 10.1145/2464996.2465009.

[84] R. Ge, X. Feng, W. Feng, and K. W. Cameron, “Cpu miser: a performance-directed, run-time system for power-aware clusters,” in *ICPP ’07 Proceedings of the 2007 International Conference on Parallel Processing*, 2007. DOI: 10.1109/ICPP.2007.29.

[85] C.-H. H. Hsu and W.-C. C. Feng, “A power-aware run-time system for high-performance computing,” in *Proceedings of the ACM/IEEE 2005 Supercomputing Conference, SC’05*, IEEE, 2005, ISBN: 1595930612. DOI: 10.1109/SC.2005.3.

[86] V. W. Freeh and D. K. Lowenthal, “Just in time dynamic voltage scaling : exploiting inter-node slack to save energy in mpi,” in *ACM/IEEE SC 2005 Conference (SC’05)*, 2005, ISBN: 1595930612. DOI: 10.1109/SC.2005.39.

[87] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, “Adagio: making dvs practical for complex hpc applications,” *Ics*, 2009. DOI: 10.1145/1542275.1542340.

[88] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "A run-time system for power-constrained hpc applications," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015, ISBN: 978-3-319-26927-6. DOI: 10.1007/978-3-319-20119-1_28.

[89] J. Eastep, S. Sylvester, C. Cantalupo, *et al.*, "Global extensible open power manager: a vehicle for hpc community collaboration on co-designed energy management solutions," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, ISBN: 9783319586663. DOI: 10.1007/978-3-319-58667-0_21.

[90] D. De Sensi, M. Torquati, and M. Danelutto, "A reconfiguration algorithm for power-aware parallel applications," *ACM Transactions on Architecture and Code Optimization*, Dec. 2016. DOI: 10.1145/3004054.

[91] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffmann, "A probabilistic graphical model-based approach for minimizing energy under performance constraints," in *ACM SIGARCH Computer Architecture News*, 2015, ISBN: 978-1-4503-2835-7. DOI: 10.1145/2786763.2694373.

[92] S. Labasan, M. Larsen, H. Childs, and B. Rountree, "Paviz: a power-adaptive framework for optimizing visualization performance," in *EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2017, ISBN: 978-3-03868-034-5. DOI: 10.2312/pgv.20171088.

[93] S. Baskiyar and R. Abdel-Kader, "Energy aware dag scheduling on heterogeneous systems," *Cluster Computing*, 2010. DOI: 10.1007/s10586-009-0119-6.

[94] M. Guzek, J. E. Pecero, B. Dorronsoro, and P. Bouvry, "Multi-objective evolutionary algorithms for energy-aware scheduling on distributed computing systems," *Applied Soft Computing*, 2014. DOI: 10.1016/j.asoc.2014.07.010.

[95] K. H. Tsoi and W. Luk, "Power profiling and optimization for heterogeneous multi-core systems," *ACM SIGARCH Computer Architecture News*, 2011. DOI: 10.1145/2082156.2082159.

[96] G. Wang and X. Ren, "Power-efficient work distribution method for cpu-gpu heterogeneous system," *International Symposium on Parallel and Distributed Processing with Applications*, 2010. DOI: 10.1109/ISPA.2010.22.

[97] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin, "Power-efficient time-sensitive mapping in heterogeneous systems," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12*, 2012, ISBN: 9781450311823. DOI: 10.1145/2370816.2370822.

[98] S. G. Johnson, *The nlopt nonlinear-optimization package*.
[Online]. Available: `http://ab-initio.mit.edu/nlopt`.